Algebraic Structures and Discrete Mathematics
Class notes for course MACS 358
Colorado School of Mines

Nicolas M. Thiéry

E-mail address: `nthiery@users.sf.net mailto:nthiery@users.sf.net`

URL: `http://www.lapcs.univ-lyon1.fr/~nthiery/`

Current address: LaPCS, Université Claude Bernard Lyon I, Bâtiment recherche [B], 50, avenue Tony-Garnier, Domaine de Gerland,, 69366 Lyon Cedex 07, FRANCE

Those notes grew up from the class MACS 358 that I taught during three semesters at the Colorado School of Mines, using the textbook *Mathematical Structures for Computer Sciences* (4th edition) by Judith L. Gersting[**1**].

Moreover, any suggestions for improvements are warmly welcomed.

I use LyX `http://www.lyx.org` to typeset them, as well as to display them in class on a LCD-projector plugged in my laptop. The dvi, PDF, ps, HTML and pure text versions are then created using LaTeX, Hevea `http://para.inria.fr/~maranget/hevea/`, Htmlpp `http://htmlpp.sourceforge.net`, and some Makefile magic. I highly recommend the PDF of ps versions, since the HTML and pure text versions are far from perfect due to the limitations of those formats.

Table des matières

CHAPITRE 1

Introduction

1.1. Course Objectives

Mathematical tools for solving problems arising from computer science.

Using a computer to solve problems.

1.2. Pólya's hints for solving a problem

There are two main kinds of problems :

(1) Problems to solve
(2) Problems to prove

Here is a list of general questions that will guide you when you try to solve a problem.

Most of them are borrowed from "How to solve it" by Pólya[3, p. XVI].

(*subliminal hint* you definitely should read this book !)

Whenever you get stuck and don't know what to do, browse through them, and most likely one of them will give you something to try out.

1.2.1. Phase 1 : Understanding the problem.

You have to understand the problem.

(1) What is the goal ?
   – What is the unknown ?
   – What are the data ?
   – Do you need all data ? Can you simplify the data ?
(2) What is the condition ?
   – Is it possible to satisfy the condition ?
   – Is the condition sufficient to determine the unknown ?
   – Or is it insufficient ? or redundant ? or contradictory ?
   – Separate the various parts of the condition. Can you write them down ?
(3) Look at examples. Draw a figure. Introduce suitable notations.
(4) How would you store the data on a computer ?

1.2.2. Phase 2 : Devising a plan.
Find the connection between the data and the unknown.

You may be obliged to consider auxiliary problems if an immediate connection cannot be found.

You should obtain eventually a *plan* of the solution.

(1) Have you seen it before ?
   - Have you seen the same problem in a slightly different form ?
   - Do you know a related problem ? Do you know a theorem that could be useful ?
   - Look at the unknown ! Try to think of a familiar problem having the same or similar unknown.

(2) Here is a problem related to yours and solved before. Could you use it ?
   - Could you use its result ? Could you use its method ?
   - Should you introduce some auxiliary element in order to make its use possible ?

(3) Could you restate the problem ? Could you restate it still differently ?

(4) Go back to definitions.

(5) If you cannot solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem ? A more general problem ? A more special problem ? An analogous problem ?
   - Could you solve a part of the problem ? Keep only a part of the condition, drop the other part. How far is the unknown then determined, how can it vary ?
   - Could you derive something useful from the data ? Could you think of other data appropriate to determine the unknown ? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other ?

(6) Did you use all the data ? Did you use the whole condition ? Have you taken into account all essential notions involved in the problems ?

1.2.3. Phase 3 : Carrying out the plan.
Carry out your plan.

(1) Carrying out your plan of the solution, *check each step*. Can you see clearly that the step is correct ? Can you prove it is correct ?

1.2.4. Phase 4 : Looking back.
Examine the solution obtained.

(1) Can you check the result ? Can you check the argument ?

(2) Can you derive the result differently ? Can you see it at a glance ?

(3) What's the structure behind ?

(4) Can you use the result, or the method for some other problem ?

(5) Can you make it an algorithm ?
   - Is your algorithm correct ?
   - Can you prove it is correct ?

1.3. Some “research” about Mazes

1.4. Conclusion

The purpose of this exercise was to browse quickly through what we are going to do this semester :

1.4.1. General problem solving techniques : We have used Pólya’s list of questions as a guide to solve our problems. We will do this over and over. Usually the main difficulty with a problem is to get started. Whenever you get stuck on a problem, and don’t know what to do, you should refer to this list, and see if some of the questions could give you something to try out.

1.4.2. Proofs. At some time, we had a solution for exiting from a maze. However, we were not sure if it worked all the time or not. Well, the only way to be sure that something is correct is to PROVE it.

We will need to learn how to prove things before anything else. In particular we will want to prove that certain algorithms are correct. That will be the core of our first month of class.

So what’s a proof ? Basically, it’s a message written by a human A to convince another human B that some fact is true (possibly A and B are the same person). When B reads through the message, he should not have any choice at the end but to say "I agree". To achieve this goal, our primary tool will be Formal Logic.

It’s not easy to write good proofs. For example, a proof should be short enough that you don’t get lost in the details, and detailed enough that you are sure not to have left a hole somewhere. Learning to write proofs is like learning to write. The only way is to write a lot of proofs yourself, and to take model on other’s proofs. We will learn this progressively.

1.4.3. Discrete structures : We have seen that the very structure of a maze (once we have removed all extraneous information like color, shape and so on) can be formalized with a graph, that is a set of nodes which are connected or not by edges.

A graph is a good example of discrete object, or structure (in opposition to a continuous object like a curve). We are going to see other discrete structures, and learn to recognize them when the arise at the very heart of problems. We are also going to see how to deal with such structures (algorithms and such).

1.4.4. Counting objects of a certain kind (Combinatorics). How many mazes ? how many ordered trees ?

1.4.5. Algebraic structures. That’s another kind of structure that can arise in our problems. Addition, multiplication and other algebraic operations are very powerful tools. We will see that such operations can often be defined for other objects that the usual integers or real number.

1.4.6. Making a solution into an algorithm, and implementing it.

Première partie

Formal Logic

CHAPITRE 2

Propositional Logic

2.1. Introduction

2.1.1. What is a proof?

DÉFINITION. A *proof* is a piece of text written by a human to convince another human that some fact is true.

PROBLEM 2.1.1. [**1**, p. 2]

You have been selected to serve on jury duty for a criminal case. The attorney for the defense argues as follow :

"If my client is guilty, then the knife was in the drawer. Either the knife was not in the drawer, or Jason Pritchard saw the knife. If the knife was not there on October 10, it follows that Jason Pritchard did not see the knife. Furthermore, if the knife was there on October 10, then the knife was in the drawer and also the hammer was in the barn. But we all know that the hammer was not in the barn. Therefore, ladies and gentlemen of the jury, my client is innocent."

Are you convinced ?

Problems :

- What is the goal ?
- What are the hypothesis ?
- What is the logical link between the statements ?
- There is plenty of unrelevant information.

A good proof consists of :

- A description of the goal
- Hypothesis
- Statements, one after the other.
  Each one derives logically from the previous ones.
- Conclusion

Let's look at a very simple example :

THÉORÈME. *Let x and y be two integers.*
If x is odd and y is odd, then xy is odd.

DÉMONSTRATION. Assume x is odd and y is odd.
Let k be an integer such that $x = 2k + 1$.
Let l be an integer such that $y = 2l + 1$.
Then, $xy = (2k + 1)(2l + 1) = 2(2kl + k + l) + 1$.
Note that $2kl + k + l$ is an integer.
Conclusion : xy is odd. $\square$

2.1.2. Formal and Informal proofs. What do we mean by "derives logically from the previous ones" ?

What operations shall we consider as legal ?

To learn more about this, we need a *model* of what we intuitively consider as true, or *logic*.

This model we are going to construct is called *formal logic*.

2.1.2.1. *Formal systems.*

Définition. A *formal system* is build by providing :
– a set of strings : the *axioms*
– a set of rewriting rules
A string that can be obtained from the axioms by applying rewriting rules is called a *theorem* of the formal system.

The *proof* of a theorem is a description of the axioms and rules used to get the theorem.

The set of all theorems is called the *language* defined by the formal system.

Exemple. The MIU formal system[**2**]

A formal system does not need to have any *meaning* or *semantic*. It's pure syntax.

We will build formal logic as a formal system, with a meaning !

– The strings will be logic formulas
– The axioms will be simple logic formulas that we interpret intuitively true
– The rewriting rules will be transformations we interpret intuitively as valid

Then, any theorem of this formal system shall be interpreted as true.

We will have to be very careful with the choice of the axioms and rules !

2.1.2.2. *Proofs and programming.* Writing a proof and a writing a program are very similar :

Proofs	Programming	Purpose
Theorem	Function prototype	How to use it
Proof	Function definition	How it works
Intuitive proof	Comments	Explanation for the reader
Informal proof	Program in a high level language	Human readable
Formal proof	Program in assembly language	Machine readable
Checking an informal proof	Compiling	Human readable -> Machine readable

This comparison is not gratuitous. There are deep links between proofs and programming which have been studied a lot in the last decades. This is called the Curry-Howard correspondence.

2.2. Statements, Symbolic Representation, and Tautologies

As a first step, let's define logic formulas and their interpretation.

2.2.1. Statements and Propositions.

DÉFINITION. A *statement* (or *proposition*) is a sentence which is either true or false.

EXEMPLE. Which sentences below are statements ?
– The sky is cloudy
– It's raining
– Are you happy ?
– He is happy
– Three divides five
– There are life forms on other planets in the universe

NOTATION 2.2.1. A, B, C : symbols representing a statement.

Each of those symbols can be given a truth value, either True (T for short) or False (F for short).

1 is a statement if truth value True.

0 is a statement if truth value False.

2.2.2. Connectives.
How can we combine two statements ?

2.2.2.1. *And (conjunction), or (disjunction).*

EXEMPLE. "The sky is cloudy" : A. "It's raining" : B.
– "The sky is cloudy AND it's raining" : $A \wedge B$
– "The sky is cloudy OR it's raining" : $A \vee B$

DÉFINITION. The *truth table* of a formula describes when it's true or not, given the truth value of the symbols.

A	B	$A \wedge B$	$A \vee B$
T	T		
T	F		
F	T		
F	F		

2.2.2.2. *Exclusive or.*

EXEMPLE. "Cheese or desert"

You get cheese : A

You get desert : B

DÉFINITION. A exclusive or B : A xor B.

Truth table :

A	B	A xor B
T	T	
T	F	
F	T	
F	F	

2.2.2.3. *Implication.*

EXEMPLE. “If I get my driver license, I will open a bottle of champagne”.

“I get my driver license” : A.

“I open a bottle of champagne” : B.

DÉFINITION. *A implies B* : $A \to B$.

Truth table :

A	B	$A \to B$
T	T	
T	F	
F	T	
F	F	

Why this should be true :

– no real meaning but we have to pickup one to make it easy
– Most reasonable is true

Why this should be false :

This one can be tricky. To help you make up your mind, here are some examples :

– If I don’t get my driver license, but still I open a bottle of champagne, am I a liar ?
– “If I open a bottle of champagne, I will get my driver license”
– “When pig flies, I’ll give you 2 billions dollars”

Well, as strange as it can sound, $F \to T$ is a true formula.

If you start from an utterly false hypothesis, you can prove anything.

“Avec des si, on pourrait mettre Paris en bouteille”

“With ifs, Paris could be put in a bottle”

REMARQUE. $A \to B$ and $B \to A$ are not the same !

Sounds stupid ?

Look carefully around you ! This is the most common logic error in peoples argumentations.

And it’s not always an involuntary error.

EXERCICE 1. Give the antecedent and the consequent in the following sentences
– If the rain continues, then the river will flood
– A sufficient condition for network failure is that the central switch goes down
– It’s raining only if there are clouds in the sky
– It’s raining if there are clouds in the sky
– It’s raining if and only if there are clouds in the sky
– The avocados are ripe only if they are dark and soft
– A good diet is a necessary condition for a healthy cat

2.2.2.4. *Equivalence.*

EXEMPLE. The knight will win if and only if his armor is strong

The knight will win : A.

His armor is strong : B.

This sentence contains two informations :
– The knight will win if his armor is strong : $A \leftarrow B$.
– The knight will win only if his armor is strong : $A \rightarrow B$.

DÉFINITION. *A is equivalent to B* : $A \leftrightarrow B$.

Truth table :

A	B	$A \leftrightarrow B$
T	T	
T	F	
F	T	
F	F	

2.2.2.5. *Not.* So far we defined binary connectives, that is connectives which took two arguments.

Here is now a unary connective, taking just one argument.

EXEMPLE. It will not rain tomorrow

The knight will win : A.

His armor is strong : B.

This sentence contains two informations :
– The knight will win if his armor is strong : $A \leftarrow B$.
– The knight will win only if his armor is strong : $A \rightarrow B$.

DÉFINITION. not A : A' (more usual notation : $\neg A$)

Truth table :

A	A'
T	
F	

2.2.3. Complex Well Formed Formulas.

DÉFINITION. A *formula* is a string of
– symbols : A, B, ...
– connectives : $\vee$, $\wedge$, $\rightarrow$, $\leftarrow$, $'$, ...
– parenthesis : (,), [,]

A *well formed formula* (or wff) is a formula which satisfies certain syntax rules.

NOTATION 2.2.2. Symbols representing wff : P, Q, R.

2.2.3.1. *Order of precedence.* The formula $A \wedge B \rightarrow C'$ is ambiguous. It can be read as :
– $(A \wedge B) \rightarrow (C')$,
– $A \wedge [B \rightarrow (C')]$,
– $[(A \wedge B) \rightarrow C]'$
Solution 1 : always put all parenthesis (not very readable).
Solution 2 : put an order of precedence between the operators :

(1) Connectives within parentheses, innermost parenthesis first
(2) $'$
(3) $\wedge$, $\vee$
(4) $\rightarrow$, $\leftarrow$
(5) $\leftrightarrow$

With this order of preference, $A \wedge B \rightarrow C'$ unambiguously refer to $(A \wedge B) \rightarrow (C)'$.

2.2.3.2. *Evaluation of a wff.* From the truth values of the basic statements, and the elementary truth tables, one can *evaluate* the truth value of a wff.

Exemple. $A \wedge B \rightarrow C'$, with A true, B false and C true :
$A \wedge B$: false
C' : false
$A \wedge B \rightarrow C'$: true

There is a recursive algorithm to compute the evaluation of a wff.

2.2.3.3. *Truth table of a wff.* This yields an algorithm for computing the truth table of a wff.

A	B	C	$A \wedge B$	C'	$(A \wedge B) \rightarrow (C')$
T	T	T			
T	T	F			
T	F	T			
T	F	F			
F	T	T			
F	T	F			
F	F	T			
F	F	F			

Problem 2.2.3. What's the size of a truth table ?

Problem 2.2.4. Can any truth table be obtained by a wff ?

2.2.4. Tautologies and Contradictions.

Définition. A *tautology* is a statement that is always true.
A c*ontradiction* is a statement that is always false.

Exemple. $A \vee A'$
"Today the sun shine or today the sun does not shine".

Exercice 2. Are the following wff tautologies or contradictions ?

(1) $A \wedge A'$

(2) $A \leftrightarrow A$

(3) $(A \wedge B) \leftrightarrow (B \wedge A)$

(4) $(A \wedge B) \rightarrow A$

Algorithm to check for a tautology ?

Compute the truth table !

It's slow (2^n, where n is the number of propositions), but it works.

For certain simple forms of propositions, there is a much faster algorithm (see [**1**, Tautology test, p. 13]).

Exemple. Is $[(A \wedge B \rightarrow C') \wedge (E \leftarrow D)] \leftrightarrow [(E \leftarrow D) \wedge (A \wedge B \rightarrow C')]$ a tautology ?

Let $P := (A \wedge B \rightarrow C')$ and $Q := (E \leftarrow D)$.

The formula above is $[P \wedge Q] \leftrightarrow [Q \wedge P]$.

Let's prove it's a tautology.

Take some truth values for A, B, C, D, E.

P and Q will then have some truth value.

But we know that $(A \wedge B) \leftrightarrow (B \wedge A)$ is a tautology.

So $[P \wedge Q] \leftrightarrow [Q \wedge P]$ will be true !

We don't actually need to compute the truth table of P and Q.

Remarque. Whenever you have some tautology, if you replace ALL occurrences of some basic statement (say A) by a more complicated wff, you still get a tautology.

2.2.5. Equivalence of wff.

Let P and Q be two wff, and suppose $P \leftrightarrow Q$ is a tautology.

Then P and Q have the same truth table.

Définition. P and Q are *equivalent* (noted $P \Leftrightarrow Q$) if they have the same truth table.

Remarque. Writing $P \leftrightarrow Q$ or $P \Leftrightarrow Q$ is different.

In the first case, you just represent a formula, which can be true or false.

In the second case, you claim that the wff $P \leftrightarrow Q$ is a tautology.

2.2.5.1. *Basic equivalent wff.*

What is the negation of "Peter is tall and thin" ?

Identity (id)	$A \vee 0 \Leftrightarrow A$ $A \wedge 1 \Leftrightarrow A$
Complement (comp)	$A \vee A' \Leftrightarrow 1$ $A \wedge A' \Leftrightarrow 0$
Commutativity (comm)	$A \vee B \Leftrightarrow B \vee A$ $A \wedge B \Leftrightarrow B \wedge A$
Associativity (ass)	$(A \vee B) \vee C \Leftrightarrow A \vee (B \vee C) \Leftrightarrow A \vee B \vee C$ $(A \wedge B) \wedge C \Leftrightarrow A \wedge (B \wedge C) \Leftrightarrow A \wedge B \wedge C$
Distributivity (dist)	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$ $A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
Double negation (dn)	$A'' \Leftrightarrow A$
De Morgan's Law (dm)	$(A \vee B)' \Leftrightarrow A' \wedge B'$ $(A \wedge B)' \Leftrightarrow A' \vee B'$
Implication (imp)	$A' \vee B \Leftrightarrow A \rightarrow B$

Remarque. Notice the duality between $\vee$ and $\wedge$.

Unlike $+$ and $\cdot$ they play a perfectly symmetric role in all formulas.

2.2.5.2. *More equivalent wff using substitution.*

Exemple. $(A \wedge B) \wedge 1 \Leftrightarrow (A \wedge B)$.

Remarque. Let P and Q be equivalent wff.

If you replace all occurrences of some basic statement by another wff, you still get two equivalent wff's.

Exemple. $A \wedge (B \vee C) \Leftrightarrow A \wedge (C \vee B)$

Remarque. Let P and Q be equivalent wff.

If P appears *exactly* as a subformula of a bigger wff, then you can substitute P by Q and get an equivalent wff.

2.2.5.3. *What's the point ?* Rewriting a formula to simplify it :

Exercice 3. [**1**, Example 7 p.12]

```
If ((outflow > inflow) and not ((outflow>inflow) and (pressure < 1000)))
  Do something ;
Else
  Do something else ;
EndIf
```

Write the wwf corresponding to the test.

Rewrite this wwf using equivalence rules.

Electronic circuits and logical gates :

Exercice 4. Build an electronic circuit for $(A \wedge B) \rightarrow C'$.
- Case 1 : you have electronic gates TRUE, FALSE, AND, OR, NAND.
- Case 2 : you only have electronic gates NAND.

2.3. Propositional logic

2.3.1. Arguments. We have seen propositions, well formed formula for formalizing statements.

We now need to formalize arguments.

Exemple. [**1**, Example 17 p. 28]

Let's formalize the following argument :

"If interest rates drop, the housing market will improve.

Either the federal discount rate will drop, or the housing market will not improve.

Interest rates will drop.

Therefore the federal discount rate will drop."

Basic statements :
- I : Interest rates drop
- H : The housing market will improve
- F : The federal discount rate will drop

Argument : $(I \to H) \wedge (F \vee H') \wedge I \;\to\; F$

Définition. An *argument* is a wff of the form $P_1 \wedge P_2 \wedge \cdots \wedge P_n \;\to\; Q$.

$P_1, \cdots, P_n$ are the *hypothesis*.

Q is the *conclusion*.

The argument is *valid* iff $P_1 \wedge P_2 \wedge \cdots \wedge P_n \;\to\; Q$ is a tautology.

Exercice 5. Prove that the following arguments are valid :

(1) $[A \to (B \to C)] \;\to\; [(A \wedge B) \to C]$
(2) $[(A \wedge B) \to C] \;\to\; [A \to (B \to C)]$

Why is this reasonable ?

"If the sun shine then if you don't put sun cream, you will get burned" : $A \;\to\; (B \to C)$

"If the sun shine and you don't put sun cream, then you will get burned" : $(A \wedge B) \;\to\; C$

Exemple. "George Washington was the first president of the United States.

Therefore everyday has 24 hours."

We put some more restrictions to only keep "meaningful" arguments.
- Q has some relation with the P_i.
- the P_i are not in contradiction

2.3.2. Proof sequences. So now, how to prove that an argument is a tautology ?

(1) Use the truth table
(2) Use formal logic !

EXEMPLE. Let's prove that the argument $A \vee (A \vee B)' \ \rightarrow\ (A \vee B')$ is valid.

We are lazy to compute the truth table.

Instead, we assume that the hypothesis is valid, and use a series of equivalences to deduce that the conclusion is valid.

$$\begin{array}{lll}
1. & A \vee (A \vee B)' & (\text{Hypothesis}) \\
2. & A \vee (A' \wedge B') & (\text{De Morgan}'s,\ 1) \\
3. & (A \vee A') \wedge (A \vee B') & (\text{Distributivity},\ 2) \\
4. & 1 \wedge (A \vee B') & (\text{Complement},\ 3) \\
5. & (A \vee B') & (\text{Identity},\ 4)
\end{array}$$

What we just did is called a *proof sequence.*

This is our first proof using *formal logic.*

EXERCICE 6. Prove that the following argument is valid :
$(A \vee B)' \vee (A' \wedge B) \ \rightarrow\ A'$

DÉFINITION. *Formal Logic* : wff + set of derivation rules (formal system).

What derivation rules shall we accept ?

We want our formal logic to be :
- Correct
- Complete
- Minimal
- Powerful

2.3.3. Looking for derivation Rules.

EXEMPLE. $A \wedge B \ \rightarrow\ A$

Equivalences rules are not enough.

We can't prove the previous statement, because $A \wedge B$ is strictly stronger than A.

We have build equivalence rules from equivalences $P \leftrightarrow Q$.

We can build *Inference rules* from any valid argument $P \rightarrow Q$.

Rule name	From	Can derive
modus ponens (mp)	P, $(P \rightarrow Q)$	Q
modus tollens (mt)	$(P \rightarrow Q)$, Q'	P'
conjunction (con)	P, Q	$P \wedge Q$
simplification (sim)	$P \wedge Q$	P, Q
addition (add)	P	$P \vee Q$

EXEMPLE. $A \wedge (A \rightarrow B) \wedge (B \rightarrow C) \ \rightarrow\ C$

$$\begin{array}{lll}
1. & A \wedge (A \rightarrow B) \wedge (B \rightarrow C) & (\text{Hypothesis}) \\
2. & A & (\text{Simplification},\ 1) \\
3. & A \rightarrow B & (\text{Simplification},\ 1) \\
4. & B & (\text{Modus ponens},\ 2,\ 3) \\
5. & B \rightarrow C & (\text{Simplification},\ 1) \\
6. & C & (\text{Modus ponens},\ 4,\ 5)
\end{array}$$

Exemple. $(A \to (B \vee C)) \wedge B' \wedge C' \ \to \ A'$

1.	$(A \to (B \vee C)) \wedge B' \wedge C'$	(Hypothesis)
2.	$(A \to (B \vee C))$	(Simplification 1)
3.	$(B' \wedge C')$	(Simplification 1)
4.	$(B \vee C)'$	(De Morgan′s 3)
5.	A'	(Modus tollens 4, 5)

Exercice 7. Prove the validity of the following arguments using formal logic :

(1) $A \wedge (B \to C) \wedge [(A \wedge B) \to (D \vee C')] \wedge B \to D$

(2) $A' \wedge (A \vee B) \to B$

(3) $A' \wedge B \wedge [B \to (A \vee C)] \to C$

(4) $A \wedge A' \to B$

Theorem 2.3.1. *This formal logic is correct and complete.*

The correctness of the formal logic comes from the way we built the rules from valid arguments.
Proving it's complete is far more difficult.

2.3.4. Methods to make it easier. Our logic is complete, but still not very practical to use.
Here are some tricks to make it easier.
Most of the time, the proofs start like this :

1.	$(A \to (B \vee C)) \wedge B' \wedge C'$	(hyp)
2.	$(A \to (B \vee C))$	(Simp 1)
3.	$(B' \wedge C')$	(Simp 1)
...	...	...

The steps 2 and 3 are pretty trivial, and it's reasonable to start directly with :

1.	$(A \to (B \vee C))$	(hyp)
2.	$(B' \wedge C')$	(hyp)
...	...	...

2.3.4.1. *The deduction method.*

Exemple. $(A \to B) \wedge (B \to C) \ \to \ (A \to C)$
The obvious start is :

1.	$(A \to B)$	(hyp)
2.	$(B \to C)$	(hyp)

But then it's pretty painful :

3.	$A' \vee B$	(imp 1)
4.	$A' \vee (B \to C)$	(add 2)
5.	$(A' \vee B) \wedge (A' \vee (B \to C))$	(con 3, 4)
6.	$A' \vee (B \wedge (B' \to C))$	(dist 5)
7.	$A' \vee C$	(mp 6)
8.	$A \to C$	(imp 7)

Here it's easier to first rewrite the argument into an equivalent argument.

We have seen that $P \to (Q \to R)$ and $(P \wedge Q) \to R$ are equivalent.

So we can prove $A \wedge (A \to B) \wedge (B \to C) \;\to\; C$ instead, which is straightforward :

1.	A	(hyp)
1.	$(A \to B)$	(hyp)
2.	$(B \to C)$	(hyp)
3.	B	(mt 1, 3)
4.	C	(mt 2, 4)

This trick is called the *deduction method*.

It's powerful because we have :

– a simpler goal
– more hypothesis

EXERCICE 8. $(A \to B) \wedge [B \to (C \to D)] \wedge [A \to (B \to C)] \;\to\; (A \to D)$

2.3.4.2. *Additional deduction rules.* As we have seen, any valid argument could be transformed into a new rule.

EXEMPLE. Here are two useful rules.

Rule name	From	Can derive
hypothetical syllogism (hs)	$P \to Q,\ Q \to R$	$P \to R$
contraposition	$P \to Q$	$Q' \to P'$

Adding those new rules won't change the power of the formal logic, but can make it more convenient. On the other hand, a minimal formal logic is nice since it's less error prone, and it's easier to remember all the rules.

It's up to you to choose which rules you want to use (and learn).

EXEMPLE. $(P \to Q) \wedge (P' \to Q) \to Q$.

2.4. Verbal arguments

We can now start checking verbal arguments.

EXEMPLE. Let's prove the following verbal argument[**1**, Example 17 p. 28] :

“If interest rates drop, the housing market will improve.

Either the federal discount rate will drop, or the housing market will not improve.

Interest rates will drop.

Therefore the federal discount rate will drop.”

EXERCICE 9. Prove the following verbal argument[**1**, Ex. 30 p. 32] :

“If the program is efficient, it executes quickly.

Either the program is efficient or it has a bug.

However, the program does not execute quickly.

Therefore it has a bug.”

2.5. Summary

2.5.1. Different levels of logic. Three levels for arguments :

– formal argument as a string of symbols, proof sequence (syntactic level).
– formal argument as a truth table (semantic level).
– verbal argument (interpretation level).

Any verbal argument based on propositions can be translated into a formal argument.

The propositional logic system is complete and correct :

– Any formal argument proved syntactically is valid semantically.
– Any semantically valid formal argument can be proved syntactically (it can be hard, though !).

2.5.2. Syntactic proof versus semantic proof. Testing an argument by computing it's truth table is purely mechanical.

So, what's the point of formal logic ?

– It's often faster and shorter to write
– You have more chance to grab the meaning of an argument
– In predicate logic we won't be able to rely on truth tables

CHAPITRE 3

Predicate Logic (sections 1.3, 1.4)

3.1. Introduction

Exemple. “Every man has a brain ; Socrate is a man ; therefore Socrate has a brain.”

Is propositional logic expressive enough for this sentence ?

What is missing ?

3.1.1. Quantifiers, Predicates, Validity.

Exemple. “Every man has a brain”

“For all man, this man has a brain”

New features : constants / variables / quantifier / predicates

Définition. Quantifiers :
- For all
- For every
- There exists
- For at least one
- For each
- For any
- For some

Universal quantifier : $(\forall x)$

Existential quantifier : $(\exists x)$

Always place the quantifiers inside ( ).

3.1.2. Predicates :

Définition. *Predicate* : a statement that describes a property of a variable

Exemple. $P(x)$: x has a brain

“For all man, this man has a brain”

Translates into $(\forall x)\ P(x)$

As in propositional logic, we can build *well formed formulas* from predicates and logic connectives.

Exemple. $(\forall x)\ B(x) \to R(x)'$: if x is blue, then x is not red.

3.1.3. Domain of interpretation : If x is a car, the sentence above $(\forall x)\ P(x)$ is false.
The truth of the sentence depends on where you take x from !

DÉFINITION. *Domain of interpretation* :
The collection of objects from which x may be selected

EXEMPLE. $(\forall x)\ (x > 0)$
DOI : all integers
DOI : all integers > 10
DOI : all humans

EXEMPLE. For any thing, if it's a man then it has a brain
$P(x)$: x is a man
$Q(x)$: x has a brain
$(\forall x)\ P(x) \rightarrow Q(x)$

EXERCICE 10. Write a formula for the following sentence :
"There exists a man which is called Socrate"

3.1.4. N-Ary predicates :

DÉFINITION. *Binary predicate* : predicate which takes two variables
N-Ary predicate : predicate which takes several variables

– $P(x, y) = x > y$ (DOI : integers)
– $P(Q, B)$: Q is the author of the book B (DOI : all authors / books)
– Everyone has exactly one best friend :
 $B(X, Y)$: Y is the best friend of X
 $(\forall x)\ (\exists y)\ B(X, Y) \wedge [\ (\forall Z)\ B(X, Z) \rightarrow (Z = Y)\ ]$
 For any person X, there exists a person Y, such that :

(1) Y is the best friend of X
(2) For any person Z, if Z is the best friend of X, then Z is actually Y.

3.1.5. Truth value. Can we define the Truth table of a wff ?

PROBLEM 3.1.1. The domain of interpretation may be infinite.

To assess the truth value, we need to know :

(1) What is the domain of interpretation.
(2) What property of elements of the DOI does P(x) represent.
(3) An assignment for each constant.

DÉFINITION. This is called an *interpretation.*

EXEMPLE. [**1**, Exercise 1 p. 41]
What is the truth value of the following wffs for this interpretation :
DOI : integers ; $O(x)$: x is odd ; $L(x)$: $x < 10$; $G(x)$: $x > 9$

(1) $(\exists x)\ O(x)$
 There exists an odd number

(2) $(\forall x)\ [L(x) \to O(x)]$

For any integer x, if x is strictly lower than 10, then x is odd ;

Any integer strictly lower than 10 is odd ;

(3) $(\exists x)\ [L(x) \wedge G(x)]$

There exists an integer x such that x is strictly lower than 10 and x is strictly larger than 9

(4) $(\forall x)\ [L(x) \vee G(x)]$

For any integer, either x is strictly lower than 10 or x is strictly bigger that 9

Exercice 11. [**1**, Exercise 3 p. 41]

Are the following wffs true for this interpretation :

DOI : states of the US.

$Q(x, y)$: x is north of y ; $P(x)$: x starts with the letter M ; a is "Mississippi".

(1) $(\forall x)\ P(x)$

(2) $(\forall x)\ (\forall y)\ (\forall z)\ [\ (Q(x, y) \wedge Q(y, z)) \ \to\ Q(x, z)\ ]$

(3) $(\exists y)\ (\exists x)\ Q(x, y)$

(4) $(\forall x)\ (\exists y)\ [P(y) \wedge Q(x, y)]$

For any state x, there exists a state y such that y starts with M and x is north of y.

(5) $(\exists y)\ Q(a, y)$

Exercice 12. Translate the following sentences into wffs, with :

DOI : world ; $D(x)$: x is a day ; $S(x)$: x is sunny ; $R(x)$: x is rainy ;

M is "Monday" ; T is "Tuesday".

(1) All days are sunny :
(2) Some days are not rainy :
(3) Every day that is sunny is not rainy :
(4) Some days are sunny and rainy :
(5) No day is both sunny and rainy :
(6) It is always a sunny day only if it is a rainy day :
(7) No day is sunny :
(8) Monday was sunny ; therefore every day will be sunny :
(9) It rained both Monday and Tuesday :
(10) If some day is rainy, then every day will be sunny :

3.1.6. Dummy variables / free variables :

Définition. A *dummy variable* is a variable which is linked to a quantifier.

The name of the variable is irrelevant.

Same thing as a local variable in a program.

Exemple. $(\exists x)\ Q(x)$ is the same wff as $(\exists z)\ Q(z)$

$(\exists x)\ (\forall y)\ Q(x, y)$ is the same wff as $(\exists z)\ (\forall t)\ Q(z, t)$ or $(\exists y)\ (\forall x)\ Q(y, x)$

Définition. A variable is *free* if it is not linked to a quantifier.

Same thing as a global variable in a program.

3.1.7. Validity.

RÉSUMÉ 3.1.2. Where are we ?

(1) We can build all the wff of predicate logic.
(2) Given a wff P, and an interpretation, we can decide the truth value of P.

DÉFINITION. *Argument* : $P_1 \wedge P_2 \wedge \cdots \wedge P_k \ \rightarrow\ Q$

– In propositional logic, an argument is *valid* if it's a tautology.
I.e. if it's true whatever truth value are assigned to each basic proposition.
– In predicate logic, an argument is *valid* if it's intrinsically true.
I.e. if it's true for ANY interpretation.

EXERCICE 13. [**1**, Exercise 16 p. 35]
Give interpretations to prove that the following wffs are not valid :

(1) $(\exists x)\ A(x) \wedge (\exists x)\ B(x) \ \rightarrow\ (\exists x)\ [\ A(x) \wedge B(x)\ ]$
(2) $(\forall x)\ (\exists y)\ P(x,y) \ \rightarrow\ (\exists x)\ (\forall y)\ P(x,y)$
(3) $(\forall x)\ [P(x) \rightarrow Q(x)] \ \rightarrow\ (\forall x)\ [P(x) \vee Q(x)]$
(4) $(\forall x)\ [\ A(x)'\ ] \ \leftrightarrow\ [(\forall x)\ A(x)\ ]'$

3.2. Predicate logic

There is an infinity of interpretations, so there is no algorithm to check the validity of a predicate.
We will have to use REASON :
– Reuse the rules from propositional logic
– Accept a few basic new rules as intuitively valid
– Use formal logic with those rules to prove arguments

3.2.1. Universal instantiation :

EXEMPLE. We want to be able to prove the following argument :
Every human is mortal ; Socrate is a man ; Therefore Socrate is mortal.
$H(x)$: x is a human ; $M(x)$: x is a mortal ; s : Socrate
$(\forall x)\ [H(x) \ \rightarrow\ M(x)] \wedge H(s) \ \rightarrow\ M(s)$

DÉFINITION. Rule of *universal instantiation* (ui) :
From : $(\forall x)\ P(x)$
Can derive : $P(s)$
Note : s can be any constant.

We decide that this rule is valid, because it is *intuitively* valid.
Proof sequence for : $(\forall x)\ [H(x) \rightarrow M(x)] \wedge H(s) \ \rightarrow\ M(s)$

1.	$(\forall x)\ [H(x) \rightarrow M(x)]$	(hyp)
2.	$H(s)$	(hyp)
3.	$H(s) \rightarrow M(s)$	(ui 1)
4.	$M(s)$	(mp 2, 3)

EXERCICE 14. Prove $(\forall x)\ [\ P(x) \rightarrow R(x)\ ] \wedge [\ R(y)'\ ] \ \rightarrow\ [\ P(y)'\ ]$

3.2.2. Universal generalization :

Exemple. "Every human is a living being. Every living being is mortal.Therefore every human is mortal."

$H(x)$: is a human

$L(x)$: is a living being

$M(x)$ is mortal

$(\forall x)\ [H(x) \to L(x)]\ \wedge\ (\forall x)\ [L(x) \to M(x)]\ \to\ (\forall x)\ [H(x) \to M(x)]$

This is clearly something we want to be able to prove, but we cannot use hypothetical syllogism directly !

Définition. Rule of *universal generalization* (ug)

From : $P(s)$

Can derive : $(\forall x)\ P(x)$

s must be an arbitrary element of the domain.

Exemple. $(\forall x)\ [H(x) \to L(x)]\ \wedge\ (\forall x)\ [L(x) \to M(x)]\ \to\ (\forall x)\ [H(x) \to M(x)]$

1.	$(\forall x)\ H(x) \to L(x)$	(hyp)
2.	$(\forall x)\ L(x) \to M(x)$	(hyp)
3.	$H(s) \to M(s)$	(ui 1)
4.	$L(s) \to M(s)$	(ui 2)
5.	$H(s) \to M(s)$	(hs 3, 4)
6.	$(\forall x)\ [H(x) \to M(x)]$	(ug 5)

Exemple. s : Socrate ; $H(x)$: x is a man ; $M(x)$: x is mortal :

1.	$M(s)$	(hyp)
2.	$H(s)' \vee M(s)$	(add 1)
3.	$H(s) \to M(s)$	(imp 2)
4.	$(\forall x)\ [H(x) \to M(x)]$	(ug 3)

Socrate is a mortal, therefore every man is a mortal.

This proof sequence is incorrect : you cannot apply ug at step 4.

Indeed s is Socrate, and not an arbitrary element of the domain.

Exercice 15. Prove the following arguments :

(1) $(\forall x)\ [P(x) \wedge Q(x)]\ \to\ (\forall x)\ P(x) \wedge (\forall x)\ Q(x)$

(2) $(\forall x)\ P(x) \wedge (\forall x)\ Q(x)\ \to\ (\forall x)\ (P(x) \wedge Q(x))$

(3) $(\forall x)\ [P(x) \wedge (\forall y)\ Q(x,y)]\ \to\ (\forall x)\ (\forall y)\ Q(x,y)$

3.2.3. Existential instantiation :

Exemple. DOI : contents of the fridge.
There exists a fruit; therefore, I can take a fruit.
$F(x)$: x is a fruit
$(\exists x)\ F(x) \ \rightarrow\ F(s)$

Définition. Rule of *existential instantiation* (ei) :
From : $(\exists x)\ P(x)$
Can derive : $P(s)$
s must be a newly created variable

Exemple. $(\forall x)\ [P(x) \rightarrow Q(x)]\ \wedge (\exists y)\ P(y)\ \rightarrow\ Q(s)$

1.	$(\forall x)\ [P(x) \rightarrow Q(x)]$	(hyp)
2.	$(\exists y)\ P(y)$	(hyp)
3.	$P(s)$	(ei 2)
4.	$P(s) \rightarrow Q(s)$	(ui 1)
5.	$Q(s)$	(mp 3, 4)

3.2.4. Existential generalization :

Exemple. DOI : contents of the fridge
I see a fruit, therefore there exists a fruit.
$F(s)\ \rightarrow\ (\exists x)\ F(x)$

Définition. Rule of *existential generalization* (eg)
From : $F(s)$
Can derive :$(\exists x)\ F(x)$

Exemple. $(\forall x)\ [P(x) \rightarrow Q(x)]\ \wedge\ (\exists y)\ P(y)\ \rightarrow\ (\exists y)\ Q(y)$

1.	$(\forall x)\ [P(x) \rightarrow Q(x)]$	(hyp)
2.	$(\exists y)\ P(y)$	(hyp)
3.	$P(s)$	(ei 2)
4.	$P(s) \rightarrow Q(s)$	(ui 1)
5.	$Q(s)$	(mp 3, 4)
6.	$(\exists y)\ Q(y)$	(eg 5)

Exemple. $(\exists x)\ P(x) \wedge (\exists x)\ Q(x)\ \rightarrow\ (\exists x)\ [P(x) \wedge Q(x)]$

1.	$(\exists x)\ P(x)$	(hyp)
2.	$(\exists x)\ Q(x)$	(hyp)
3.	$P(s)$	(ei 1)
4.	$Q(s)$	(ei 2)
5.	$P(s) \wedge Q(s)$	(add 3, 4)
6.	$(\exists x)\ [P(x) \wedge Q(x)]$	(eg 5)

We have seen that the previous argument is incorrect :
So, where is the flaw in the following proof?
For example, s cannot have been created by existential instantiation

Exercice 16. Prove or disprove the following arguments

(1) $(\exists x)\ (\forall y)\ Q(x,y)\ \rightarrow\ (\forall y)\ (\exists x)\ Q(x,y)$

(2) $(\forall x)\ (\exists y)\ Q(x,y)\ \rightarrow\ (\exists y)\ (\forall x)\ Q(x,y)$

3.2.5. Deduction Method and Temporary Hypothesis :

Exemple. $P(s) \rightarrow (\forall y)\ Q(x,y)\ \rightarrow\ (\forall y)\ [P(s) \rightarrow Q(x,y)]$

3.3. Conclusion

Goal : formalization of arguments and proofs.

Propositional logic ::

propositions A, B, ...
connectives, well formed formula
truth table
argument valid iff wff is a tautology
proofs :
– compute the truth table (algorithm)
– formal logic, deduction rules

Ex 24, 37

Predicate logic:

variables, predicates
domain of interpretation
connectives, wff
interpretation
truth table : all possible interpretations (infinite)
argument valid iff wff is true for all possible interpretations
proofs :
– no algorithm
– formal logic, deduction rules

Those two logic are correct and complete.
They are still not powerful enough to represent all real life argument.
For this we would need of more powerful logics (2nd order), that allows for quantifying other sets.
Problem : it's often difficult, if not impossible to prove that those logics are complete and correct !
We won't need to go into those details.
We have seen enough low-level logic to help us write safely less formal proofs.

Deuxième partie

Proofs

CHAPITRE 4

Proof techniques (section 2.1)

4.1. Theorems and Informal proofs

What we have seen so far :

Argument: $P_1 \wedge \cdots \wedge P_n \ \rightarrow \ Q$

Syntax: how it's written

Semantic: meaning in a given interpretation

Valid argument:

True for all interpretations

True because of its very structure

Proofs of valid argument are purely based on syntactical rewriting rules.

Only arguments that are true for all interpretations can be proved.

4.1.1. Theorems.

We are now interested to work in a particular subject (say integer arithmetic).

Définition. A *theorem* is an argument that is valid in this particular subject.

Exemple. If x is even and y is even then xy is even.

$E(x)$: x is even

$(\forall x)\ (\forall y)\ [E(x) \wedge E(y) \ \rightarrow \ E(xy)]$

This argument is true in this context, but not universally true.

How to prove it ?

Implicitly, we add new hypothesis which reflects basic facts of this subject.

For example, we add the following hypothesis :

– x is even if and only if there exists y such that $x = 2y$:

$$(\forall x)\ [P(x) \ \leftrightarrow \ (\exists y)\ x = 2y]$$

Using those new hypothesis, it's now possible to write a formal logic proof of the theorem.

See [**1**, Example 4 p. 87]

4.1.2. Formal and informal proofs : Remember that the goal of a proof is to be read by humans (in particular yourself) in order to convince those humans.

The formal proof above is convincing in the sense that you can check that all the steps are valid. However, it's very difficult to extract the meaning of the proof :

- Why does it work ?
- How could I reuse it for a similar problem ?

The problem is that the keys of the proofs are buried under layers of insignificant details.

So, starting from now, we will write *informal proofs* :

DÉFINITION. An *informal proof* is a narrative descriptions, of the ideas and of the important steps of the proof, without extraneous details.

EXEMPLE. A proof of the theorem above could be written as follow :

DÉMONSTRATION. Let x and y be two even integers. We can take n and m such that $x = 2n$ and $y = 2m$.
Then, $xy = 2(2nm)$. Since $2nm$ is an integer, xy is an even number. □

Let see which details we omitted :

(1) Let x and y be two even integers.
  Implicit universal instantiation
(2) We can take n and m such that $x = 2n$ and $y = 2m$.
  Implicit universal instantiation for x and y of the definition of an even number
  Implicit existential instantiation for n and m
(3) Then, $xy = 2(2nm)$
  Implicit use of rules of arithmetic
(4) Since $2nm$ is an integer, xy is an even number :
  Implicit universal instantiation of the definition of an even number
  Implicit universal generalization to get the final result

PROBLEM 4.1.1. Which details can we omit, and which not ?

A reader will be convinced by the proof if he can check that he could translate each step of the proof into one or several steps of a formal proof.

So, this all depends on WHO reads the proof !

You should not write your proof for your instructor, but for yourself, and for everybody else in the class.

A good rule of thumb is to imagine yourself rereading the proof in a few month, and to check that even then, you could possibly translate the proof into a formal one.

The difference between formal and informal proofs is very similar to the difference between assembly and, say, C++ or Java. A C++ program is not usable by itself. It's usable because it's possible to translate it into assembly language.

However, there does not exists, and most likely will never exists, a compiler that transforms informal proofs into formal proofs. English is much to rich a language for this.

4.1.3. Formal and informal theorems :

Most of the time, we also won't write theorems as a formal argument, but rather with an English sentence that could be translated into a formal argument :

THÉORÈME. *Let x and y be two integers. [Some definitions : interpretation]*
Assume x and y are even. [Some hypothesis : $P_1 \wedge \cdots \wedge P_n$]
Then, xy is even. [Consequent : Q]

4.1.4. To Prove or not to prove.

In textbooks, you can be asked : prove that ...
– you know in advance it's true ;
– you just have to figure out how to prove it.

Usually, in real life, you first have to find what to prove.

– you don't even know in advance if it's true.

Two jobs :

– Find the good questions
– Prove or disprove those questions

DÉFINITION. A *conjecture* is a statement that you guess is true, but that you have not proved or disproved yet.

Classical steps :

(1) Explore some examples
(2) Try to see some pattern emerging
(3) Formulate a conjecture
(4) Try to prove (or disprove it)

Steps 1-3 are inductive reasoning, whereas step 4 is deductive reasoning.

Finding the good questions is as important as solving them !

EXEMPLE. Fermat's conjecture.

4.1.5. Some "research" around the factorial.

DÉFINITION. Let n be an integer. The factorial of n is the number $n! := n(n-1)\cdots 1$.

For example, $1! = 1$ and $4! = \dot{4} \cdot 3 \cdot 2 \cdot 1 = 24$.

PROBLEM 4.1.2. How big is $n!$?

4.2. Proof techniques

We want to prove some argument of the form $P \rightarrow Q$.

4.2.1. Disproof by counter example.

CONJECTURE. *If n is a positive integer, then $n! < n^3$.*

4.2.2. Direct proof.

DÉFINITION. *Direct proof*

(1) *Assume P*
(2) Deduce Q

EXEMPLE. Prove that if x and y are even, then xy is even.

EXERCICE 17. Prove that if x and y are even, then $x + y$ is even.

4.2.3. Proof by contraposition.

EXEMPLE. Prove that if n^2 is odd, then n is odd.

Hint : prove instead that if n is even, then n^2 is even.

DÉFINITION. *Proof by contraposition :*

(1) Assume Q' (the consequent is false)
(2) Prove P' (the antecedent is also false)

This technique relies on the fact that $P \rightarrow Q$ is equivalent to $Q' \rightarrow P'$.

EXERCICE 18. Prove that xy is odd if and only if x and y are odd.

4.2.4. Exhaustive proof.

PROBLEM 4.2.1. Can a proof by example be valid ?

Yes, if there is a finite number of cases to be treated.

EXEMPLE. Propositional logic formula (the truth table is finite)

DÉFINITION. *Proof by exhaustion* means that all cases have been exhausted.
(and so are you ...).

EXEMPLE. Drawing a figure without lifting the pencil and without retracing a line.

4.2.5. Some "research" around rational numbers.

DÉFINITION. A number x is *rational* if it can be written as $\frac{p}{q}$, where p and q are integers.

EXEMPLE. $5, \frac{-7}{5}, \frac{1}{-3}, \frac{14}{4}, \frac{7}{2}, \ldots$ are rational.

PROBLEM 4.2.2. Properties of rational numbers ?

REMARQUE. If x is rational, it's always possible to choose p and q so that :
– q is positive
– p and q are *relatively prime*
I.e., the biggest common divisor of p and q is 1.

PROBLEM 4.2.3. Are all numbers rational ?

PROBLEM 4.2.4. Assume x and y are rational.

(1) Is $x + y$ rational ?
(2) Is xy rational ?
(3) Is $\frac{x}{y}$ rational ?

PROBLEM 4.2.5. Is the square root of an integer a rational number ?

PROBLEM 4.2.6. Prove that the square root of 2 is irrational.

4.2.6. Proof by contradiction.

THÉORÈME. *The square root of* 2 *is irrational.*

DÉMONSTRATION. Let's assume the square root of 2 is rational.
Let p and q be two integers such that $\sqrt{2} = \frac{p}{q}$ and p and q are relatively prime.
Then, we have $2 = \left(\frac{p}{q}\right)^2$, and so $2q^2 = p^2$.
Therefore p^2 is even, and we have seen that this implies that p is also even.
Let k be the integer such that $p = 2k$.
Then, we have $2q^2 = p^2 = (2k)^2 = 4k^2$, and so $q^2 = 2k^2$.
It follows that q^2 is even, and so q is also even.
Conclusion : p and q are both even.
That's a contradiction, since p and q are relatively prime! □

DÉFINITION. *Proof by contradiction*

(1) Assume the contrary
(2) Deduce a contradiction

This technique relies on the fact that :

(1) $Q \wedge Q'$ is always false
(2) if $P \rightarrow 0$ is true, then P is false.

4.2.7. Serendipity.

EXEMPLE. The chess board problem.

4.3. Summary

Goal	Technique	Name
$P \rightarrow Q$	Assume P; deduce Q.	Direct proof/Deduction method
$P' \rightarrow Q'$	Prove $Q \rightarrow P$.	Proof by contraposition
Q'	Assume Q; deduce a contradiction.	Proof by contradiction
$P \leftrightarrow Q$	Prove $P \rightarrow Q$; prove $Q \rightarrow P$.	
$P \wedge Q$	Prove P; prove Q.	
$P \vee Q$	Prove $P' \rightarrow Q$	
$(P_1 \vee P_2) \rightarrow Q$	Prove $P_1 \rightarrow Q$; prove $P_2 \rightarrow Q$	Proof by cases
$(\forall x)\ Q(x)$	Let x; prove $Q(x)$	ui / ug
$(\exists x)\ Q(x)$	Construct a such that $Q(a)$	eg
	Be smart	Serendipity

CHAPITRE 5

Induction - Recursion

5.1. Introduction

Définition. A *sequence* S is a list of objects, enumerated in some order :

$$S(1), S(2), S(3), \dots, S(k), \dots$$

A sequence can be defined by giving the value of $S(k)$, for all k.

Exemple. $S(k) := 2^k$ defines the sequence : $2, 4, 8, 16, 32, \dots$

Imagine instead that I give you the following recipe :
- (a) $S(1) := 1$.
- (b) If $k > 1$, then $S(k) := S(k-1) + k$.

Can you compute $S(2)$? $S(3)$? $S(4)$?

Could you compute any $S(k)$, where $k > 0$.

Proposition. *The sequence $S(k)$ is fully and uniquely defined by (a) and (b).*

Définition. We say that S is defined *recursively* by (a) and (b).
- (a) is the *base case*
- (b) is the *induction step*

Exemple. The stair and the baby.

This is the idea of recursion (or induction), which is very useful in maths and computer science.

It allows to reduce an infinite process to a finite (small) number of steps.

Exemple. Define $S(k)$ by $S(1) := 4$.

Problem : how to compute S(2) ?

Exemple. Define S(k) by $S(k) := 2S(k-1)$

Problem : both the following sequences satisfies this definition !
- $1, 2, 4, 8, 16, \dots$
- $3, 6, 12, 24, 48, \dots$

Exemple. Proof that any integer is even.

Pitfall : Both base case and induction step are necessary !

You need to know how to start, and you need to know how to continue.

5.2. Proofs by induction

PROBLEM 5.2.1. Let S be the sequence defined by :
– $S(1) = 1$
– $S(k) := S(k-1) + 2^{k-1}$
Goal : compute $S(60)$

$$\begin{array}{rclcl}
S(1) & = & 1 & = & \\
S(2) & = & 1+2 & = & \\
S(3) & = & & = & \\
S(4) & = & & = & \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
S(k) & = & 1+\cdots+2^k & = &
\end{array}$$

CONJECTURE. $S(60) =$

The formula $S(k) = 2^k - 1$ is called a *closed form formula* for $S(k)$.
It allows to compute $S(k)$ easily, without computing all the previous values $S(1), S(2), \dots S(k-1)$.
So, how to prove this conjecture ?

***Introduce some notation*:**
Let $P(k)$ be the predicate : $S(k) = 2^k - 1$.
For any positive integer k, $P(k)$ is either true or false.

***Look at examples*:**

EXERCICE 19. Try the following :

– Prove $P(1), P(2), P(3)$
– Assume that $P(27)$ is true. Can you prove that $P(28)$ is true ?
Now, can you prove $P(60)$?

5.2.1. First principle of mathematical induction :

THÉORÈME. *First principle of mathematical induction*
Let $P(k)$ be a predicate. If :
(a) $P(1)$ is true
(b) For any $k > 1$, $P(k-1)$ true implies $P(k)$ true
Then : $P(k)$ is true for all $k \geq 1$.

DÉFINITION. P is the *inductive hypothesis.*
(a) is the *base case.*
(b) is the *induction step.*

EXEMPLE. Consider the sequence S defined as above by :
(a) $S(1) := 1$.
(b) $S(k) := S(k) + 2^{k-1}$.
Let's prove that for all k, $S(k) = 2^k - 1$.

DÉMONSTRATION. Let $P(k)$ be the predicate $S(k) = 2^k - 1$.

(1) Base case :

$S(1) = 1 = 2^1 - 1$. So, $P(1)$ is true.

(2) Induction step :

Let $k > 1$ be an integer, and assume $P(k-1)$ is true : $S(k-1) = 2^{k-1} - 1$.

Then, $S(k) = S(k-1) + 2^{k-1} = 2^{k-1} - 1 + 2^{k-1} = 2^k - 1$.

So, P(k) is also true.

Conclusion : by the first principle of mathematical induction, $P(k)$ is true for all $k \geq 1$, i.e. $S(k) = 2^{k-1}$.

□

Exercice 20. Let $S(k) := 1 + 3 + 5 + \cdots + (2k-1)$. Find a closed form formula for $S(k)$.

Exercice 21. Prove that $k! > 2^k$ for any $k \geq 4$.

Exercice 22. Prove that for any integer k, $2^{2k} - 1$ is divisible by 3.

Exercice 23. Prove that $a + ar + ar^2 + \cdots + ar^n = \frac{a-ar^n}{1-r}$.

Exercice 24. Find a closed form formula for $S(k) := 1^4 + 2^4 + \cdots k^4$.
Hint : the difficulty is to find a suggestion. What tool could you use for this ?

Exercice 25. The chess board problem.

5.2.2. Other forms of induction :

Exemple. Define the sequence $F(n)$ by :
$F(1) := 1$
$F(2) := 1$
$F(k) = F(k-1) + F(k-2)$, for all $k > 2$.

Exercice 26. Can you compute $F(3), F(4), F(5)$?

This sequence is the famous and useful *Fibonacci* sequence.

Problem 5.2.2. To compute $F(k)$, you not only need $F(k-1)$, but also $F(k-2)$. So that does not fit into the previous induction scheme.

That's fine, because to compute $F(k)$, you only need to know the values of $F(r)$ for $r < k$.

Théorème. *Second principle of Mathematical induction :*
Let $P(k)$ be a predicate. If
(a) $P(1)$ is true,
(b) For any $k > 1$, [$P(r)$ true for any r, $1 \leq r < k$] implies $P(k)$ true,
then : $P(k)$ is true for all $k \geq 1$.

Exemple. Any integer is a product of prime integers

Exemple. The coin problem

We did not prove that the first (or the second) principle of induction were valid.
Let's just give an idea of the proof :

Théorème. *The three following principles are in fact equivalent :*

(1) *the first principle of induction*
(2) *(b) the second principle of induction*
(3) *(c) the principle of well ordering :*
every non empty collection of positive integers has a smallest member.

Problem 5.2.3. Is the principle of well ordering valid for $\mathbb{Z}$? $\mathbb{R}$? $\mathbb{C}$?

5.3. Other uses of induction

Induction is a much more general problem solving principle.
If you have a family of problems such that :
– There is some measure of the size of a problem
– You can solve the smallest problems
– You can breakup each bigger problems in one (or several) strictly smaller cases, and build from it a solution for the big problem.
Then, you can solve by induction any problem in your family.

Exemple. Problem : computation of $F(k)$.
Measure of the problem : k.
Value of $F(1)$ is known.
$F(k)$ can be computed from the values of $F(k-1)$ and $F(k-2)$

Exemple. Inductive definition of well formed formulas from formal logic
<basic proposition> : $A \mid B \mid C \mid \dots$
<binary connective> : $\wedge \mid \vee \mid \rightarrow \mid \leftarrow \mid \leftrightarrow$
<wff> : <basic proposition> | <wff>′ | (<wff>) | <wff> <binary connective> <wff>

This kind of inductive description is called *BNF* (Backus Naur form).

Exemple. Recursive proof of a property of wff

Théorème. *If a wff contains n propositions (counted with repetition : in $A \vee A$, there are two propositions), then it contains $n-1$ binary connectives.*

Exemple. Recursive algorithm for the computation of the value of a wff :

```
bool function Evaluation($P$, $C$)
// Precondition : $P$ is a wff, $C$ is the context, i.e.
// the truth values of all the basic propositions.
// Postcondition : returns the truth value of $P$ in the // context $C$
Begin
  If $P$ is a proposition (say $A$) Then
    Return the truth value of $A$ ;
  ElseIf $P$ is of the form ($Q$) Then
    Return Evaluation($Q$, $C$) ;
  ElseIf $P$ is of the form $Q'$ Then
```

```
          Return not Evaluation($Q$,$C$) ;
        ElseIf $P$ is of the form $Q_1 \wedge Q_2$ Then
          Return Evaluation($Q_1$, $C$)and Evaluation($Q_2$, $C$) ;
        ... // Other binary operators
        Else
          Error $P$ is not a wff ;
      End ;
```

REMARQUE. There are programming languages that are particularly well adapted to write this kind of algorithms. Actually, the real program would almost look like the pseudo-code. See for example CAML `ftp://ftp.inria.fr/lang/caml-light/`.

EXEMPLE. Bijection between words of parenthesis and forests

5.4. Conclusion

Induction is a fundamental technique for solving problems, in particular in computer science.

It's the mathematical formalization of the divide and conquer approach.

CHAPITRE 6

Proof Of Correctness (Sections 1.6, 2.3)

6.1. Introduction

"Beware of bugs in the above code ; I have only proved it correct, not tried it."

Donald E. Knuth `http://www-cs-faculty.stanford.edu/~knuth/`

How to check if a program is correct ?

Why checking if a program is correct ?

What is a correct program ?

DÉFINITION. *Specification* : precise description of how a program should behave.

A program is *correct*, if it behaves in accordance with its specifications.

How to check if a program is correct ?
– Testing
    Designing good test data sets
    You can prove there is a bug. No proof of no bug !
– Proof of Correctness

6.2. Correctness

6.2.1. Assertions.

```
...
x=sqrt(r) ;  // r should be >= 0 !
x=a[i] ;     // i should be in the correct range
...
```

DÉFINITION. *Assertion* : condition on the variables of a program that should be verified at some given step, if the program is running correctly.

Using assert in C/C++ :

```
#include <assert.h>
  ...
  assert(r>=0) ;
  x=sqrt(r) ;
  assert((0<=i) && (i<MAX)) ;
  x=a[i] ;
  ...
```

Each assertion is automatically checked :

– If the assertion is not verified, the program stops immediately with a message like :

```
Assertion failed, line 35 of file foo.c
```

– That's more informative for the user than a segmentation fault at the following line.
– This can be deactivated by adding a #define NDEBUG preprocessor directive.

With C++, java and other languages, you can use *exceptions* instead of assert :

– More informative error messages
– Error recovery mechanisms

Static type checking also makes some kind of assertion checking.

One way or the other, *check assertions in your programs.*

6.2.2. Specification of a function :

```
int factorial(int n) ;
// Preconditions : n is a non-negative integer
// Postcondition : returns n !
...
int factorial(int n) {
  assert(n>=0) ;
  if (n==0) return 1 ;
  return n*factorial(n-1) ;
}
```

The specification of a program can be formalized as follow :

– P : program
– X : input
– $P(X)$: output
– Q : precondition : predicate $Q(X)$ on the input
– R : postcondition : predicate $R(X, P(X))$ on the input and the output

EXEMPLE. Program to compute of a square root

– P : $y :=$ sqrt(x);
– X : x
– $P(x)$: y
– $Q(x)$: $x \geq 0$
– $R(x, y)$: $y^2 = x$

DÉFINITION. P is *correct* if $(\forall X)\ Q(X) \to R(X, P(X))$.

A *Hoare triple* is just a short hand notation : $\{Q\}\ P\ \{R\}$

EXEMPLE. $\{x \geq 0\}\ y :=$ sqrt(x); ; $\{y^2 = x\}$

6.3. Proof of Correctness

6.3.1. Divide and Conquer. Let P be a big program :

s_0
s_1
s_2
$\ldots$
s_{n-1}

To prove $\{Q\}$ P $\{R\}$, we break P into elementary steps, and insert assertions that describes the state of the program at each step :

$\{Q\}$
s_0
$\{R_1\}$
s_1
$\{R_1\}$
s_2
$\ldots$
s_{n-1}
$\{R\}$

P is correct, if each step is correct, i.e. the following Hoare triples hold :

– $\{Q\}$ s_0 $\{R_1\}$,
– $\{R_1\}$ s_1 $\{R_2\}$,
– ...
– $\{R_{n-1}\}$ s_{n-1} $\{R\}$

To prove that the elementary steps are correct, we will use some syntactic rules, exactly as we did in formal logic.

6.3.2. Assignment rule : Consider the following Hoare triple : $\{Q\}$ $x := e$ $\{R\}$

THÉORÈME. *If from Q you can derive R with x substituted everywhere by e, then the Hoare triple is valid.*

EXEMPLE. $\{x = 2\}$ $y := x + 1$ $\{y = 3\}$

When substituting, $y = 3$ becomes $x + 1 = 3$.

From $x = 2$, you can deduce $x + 1 = 3$.

So this Hoare triple holds.

EXEMPLE. $\{x > 0\}$ $x := x + 1$ $\{x > 1\}$

Here it can be confusing.

The same name x stands for both the value of x before and after the assignments.

If you get confused, just rename the variable :

$\{x_0 > 0\}$ $x_1 := x_0 + 1$ $\{x_1 > 1\}$

When substituting, $x_1 > 1$ becomes $x_0 + 1 > 1$, which you can deduce from $x_0 > 0$.

6.3.3. Conditional Rule.
Consider the following Hoare triple :

```
{$Q$}
if condition $B$ then
  $P_1$
else
  $P_2$
end if
{$R$}
```

THÉORÈME. *If the Hoare triples $\{Q \wedge B\}$ $P_1 \{R\}$ and $\{Q \wedge B'\}$ P_2 $\{R\}$ hold, then the Hoare triple above holds.*

EXEMPLE. [**1**, Exercise 11 p. 78]

6.3.4. Loop Rule.
Consider the following Hoare triple :

```
{$Q$}
while condition $B$ do
  $P$
end_while
{$R$}
```

$\{Q\}$ describes the state of the program before the loop.
We need to have a predicate which describes the state of the program DURING the loop :
The *loop invariant* $\{S\}$
Most of the time, $\{Q\}$ will do the job, but not always.

THÉORÈME. *If the Hoare triple $\{S \wedge B\}$ P $\{S\}$ holds, then the following Hoare triple will hold :*

```
{$S$}
while condition $B$ do
  $P$
end_while
{$S \wedge B'$}
```

6.3.5. A simple example.
Consider this little program :

```
Product(x,y)
// Precondition : x and y are non-negative integers
// Postcondition : returns the product of x and y
begin
  i :=0 ;
  j :=0 ;
  while ( (i=x)' ) do
    j=j+y ;
    i :=i+1 ;
  end while
  // Assertion : { j = x*y }
  return(j) ;
end ;
```

It's pretty clear that this algorithm it's correct.
Let's check it anyway to see how it works.

Démonstration. We need a loop invariant, which will describe the state of the program after k iterations.
A reasonable guess is $S : j = iy$
Let i_k, j_k and S_k be the value of i, j and S after k iterations.
Base case : using the assignment rule, $i_0 = 0$ and $j_0 = 0$ so S_0 holds.
Induction step :
Assume the invariant holds after $k-1$ iterations : $j_{k-1} = i_{k-1}y$
We need to prove that the invariant is preserved by the k th iteration.
Otherwise said, we have to check that the following Hoare triple holds :

```
{$S_{k-1} \wedge (i = x)'$}
 $j_k := j_{k-1} + y$ ;
 $i_k := i_{k-1} + 1$ ;
{$S_k$}
```

Let's applying the assignment rule.
By substituting i_k and j_k by the expressions they have been assigned with, we get :

$$\begin{array}{llcll} S_k & \Leftrightarrow j_k & = & i_k y & \\ & \Leftrightarrow j_{k-1}+y & = & (i_{k-1}+1)y & \text{(By the substitution rule)} \\ & \Leftrightarrow i_{k-1}y+y & = & (i_{k-1}+1)y & \text{(By } S_{k-1}\text{)} \\ & \Leftrightarrow i_{k-1}y+y & = & i_{k-1}y+y & \end{array}$$

So, S_k indeed holds.
By induction, after any number k of iterations, S_k still hold.
At the end, both S_k and $i = x$ hold, so, as expected : $j = iy = xy$ □

Remarque. We have proved that after the end of the execution the result is correct.
But what if the program does not terminate and loop forever ?
This is not usually consider a proper behavior.
We only have spoken about *partial-correctness.*
A full proof of correctness needs also a proof of termination !

6.3.6. A more sophisticated example : The Euclidean algorithm.

```
GCD(a,b)
Begin
// Precondition : a and b are non-negative integers
// with a>=b.
// Postcondition :
// returns gcd(a,b), the greater common divisor of
// a and b.
Local variables : i, j, q, r
  i :=a ;
  j :=b ;
  while j>0 do
    // Assertion :
      // r is the rest of the integer division i/j
      r := i mod j ;
      i := j ;
      j := r ;
    end while
    // Assertion : i is the gcd of a and b.
    return(i) ;
  end ;
```

Exercice 27. Use the Euclidean algorithm to compute :
`GCD(8,8)`, `GCD(14,4)`, `GCD(31,17)`.

Here the proof of correctness of the algorithm is non-trivial.

Démonstration. Let i_k and j_k be the value of i and j after k iterations.
We need to find an invariant which describes the state of the program after each iteration.
Take S_k : $\gcd(i_k,\ j_k) = \gcd(a, b)$.

(1) Base case :
    Before the loop, $i_0 = a$ and $j_0 = b$.
    So the invariant S_0 holds : $\gcd(i_0,\ j_0) = \gcd(a, b)$;
(2) Induction step :
    Assume S_{k-1} holds, that is $\gcd(i_{k-1},\ j_{k-1}) = \gcd(a, b)$.
    We just need to prove that $\gcd(i_k,\ j_k) = \gcd(i_{k-1},\ j_{k-1})$.
    By the assignment rule, we have :
    - r is the rest of the division of i_{k-1} by j_{k-1},
    - $i_k = j_{k-1}$,
    - $j_k = r$.

    So, by the definition of the integer division, we have :

$$i_{k-1} = qj_{k-1} + j_k$$

    for some integer q.

    (a) Assume x divides both i_k and j_k.
        Then, of course j_{k-1}, since $j_{k-1} = i_k$.
        By the equation above, x also divides i_{k-1}.
    (b) Assume x divides both i_{k-1} and j_{k-1}.
        Then of course x divides i_k.
        Once again, using the equation above, x also divides j_k

    Therefore, $\gcd(i_k,\ j_k) = \gcd(i_{k-1},\ j_{k-1}) = \gcd(a, b)$, as wanted.
(3) At the end :
    By induction, the loop invariant S still holds : $\gcd(i,\ j) = \gcd(a, b)$
    Moreover, the loop condition failed : $j = 0$.
    So, $\gcd(a, b) = \gcd(i,\ j) = \gcd(i,\ 0) = i$, as expected.

□

6.4. Conclusion

6.4.1. A note on automatic program proving :

- There is no mechanical way to decide even if a program terminates or not.
- Traditional imperative languages :
  - Automatic proving is very difficult because of *side effects*
  - side effects needs to be considered as some form of output
- Functional languages (lisp, scheme, ML, caml) :
  - Designed to make it easier to prove correctness
  - Fun to program. Try them !
- Two difficulties :
  - Finding the correct assertions
  - Checking that the implications holds
- Mixed approach :
  - The programmer gives the main assertions
  - The prover derives the other assertions

6.4.2. Check assertions in your programs !

- Use static type checking / assert / exceptions
- When debugging, you can ask for all assertions to be checked
- Documentation
- Help for the proof

CHAPITRE 7

Analysis of Algorithm (Section 2.5)

EXEMPLE. Finding an element in a list.

Computing the nth Fibonacci number $F(n)$.

PROBLEM 7.0.1. How long will my program take to execute ?

How much memory do I need ?

How efficient is my algorithm ?

Is it the best one ?

Which algorithm should I choose ?

Goal : "measure" the quality of an algorithm :
- Time
- Memory
- Easiness of writing

7.1. Complexity of an algorithm : Measuring time

DÉFINITION. Decide which operations are basic (i.e. multiplications).

Complexity of the algorithm : how many basic operations are needed.

EXEMPLE. Sequential search

```
SequentialSearch(x,l)
// Preconditions :
// - l is a list ["bonjour","car","hello","juggler"]
// - x is a string
// Postcondition : return true iff x is in the list
Begin
  // compare successively x with l[1], l[2], ..., l[n]
  For i From 1 To n Do
    If (x=l[i]) Then Return TRUE ; EndIf
  EndFor ;
  Return FALSE ;
End ;
```

Basic operations : comparison of two strings.

How many operations ?
- x = "unicycle" :
- x = "juggler" :
- x = "hello" :
- x = "car" :

Worst case : n basic operations

Average case : depends on the probability for x to be in the list. Difficult !

Résumé 7.1.1. n : size of the data (here, the length of the list)

Worst case : maximum number of operations for a data of size n.

Average case : average number of operations for a data of size n (difficult)

Exemple. Binary search

A non sorted dictionary is useless ! ! !

```
BinarySearch(x, l)
// Preconditions :
// - l is sorted ["bonjour","car","hello","juggler"]
// - x is a string
// Postcondition : return true iff x is in the list
Begin
  If n=1 Then
    Return x=l[1] ;
  ElseIf x <= l[n/2] Then
    Return BinarySearch(x, [l[1], ..., l[n/2]]) ;
  Else
    Return BinarySearch(x, [l[n/2]+1, ..., l[n]]) ;
  End if
End
```

Basic operation : comparison of two strings.

How many operations ?

n	operations
1	
2	
4	
8	
16	

For n, there is at most k operations, where k is the smallest integer such that $2^{k-1} \geq n$.

$\log(n)$: logarithm of n other base 2

Number of operations : $\log(n)$

7.1.1. How precise ? The estimate above does not give a precise measure of the time needed.

It depends in particular on :

- What computer you are using
- How long are the strings
- The computer language :
    - How fast are strings
    - How fast are for loops
    - How fast are function calls

A precise measure of the time is often hard to obtain.

Most of the time, to decide between to algorithms, a rough order of magnitude is enough.

Draw the graphs to get an idea if there are big differences

- $4n$ operations / $7n$ operations ?
- $4n$ operations / $7n$ operations ?
- n operations / $n+3$ operations ?
- $log(n)$ operations / n operations ?
- n operations / n^2 operations ?
- n operations / $\exp(n)$ operations ?

DÉFINITION. An algorithm is of complexity $O(n^5)$ if it needs less than $Cn^5 + K$ operations to execute, where C and K are some constant.

- $O(1)$: constant (random access to an element of an array)
- $O(\log(n))$: logarithmic (random access in a sorted list by binary search)
- $O(n)$: linear (random access in a linked list ; sequential search)
- $O(n\log(n))$: (sorting a list by a good algorithm)
- $O(n^2)$: quadratic (sorting a list by bubble sort)
- $O(n^k)$: polynomial
- $O(\exp(n))$: exponential

Worst case analysis is usually easier. It is important if your need a warranty on the time needed for each data (e.g. : real time data acquisition).

Average case analysis is more difficult. It is important when running over a large sample of data.

7.1.2. Case study : the Fibonacci sequence. Here is a program written in MuPAD `www.mupad.de` (free Computer Algebra system).

It present different implementations of the computation of the Fibonacci sequence.

```
/*
fibonacci_rec           (n : Dom : :Integer) ;
fibonacci_rec_remember  (n : Dom : :Integer) ;
fibonacci_loop_array    (n : Dom : :Integer) ;
fibonacci_loop          (n : Dom : :Integer) ;
Precondition : n non negative integer
Postcondition : returns the nth fibonacci number
Also increments the global variable operations.
*/
fibonacci_rec :=
proc(n : Dom : :Integer)
begin
  if n=0 then return(1) ; end_if ;
  if n=1 then return(1) ; end_if ;
  operations :=operations+1 ; // Count operations
  return(fibonacci_rec(n-1)+fibonacci_rec(n-2)) ;
end_proc :

fibonacci_array :=
proc(n :Dom : :Integer)
  local i, F ;
begin
  if n=0 then return(0) ; end_if ;
  if n=1 then return(1) ; end_if ;
  F[0] :=0 ;
  F[1] :=1 ;
  for i from 2 to n do
    operations := operations+1 ; // Count operations
    F[i] := F[i-1] + F[i-2] ;
  end_for ;
  return(tableau(n)) ;
end_proc :

fibonacci_loop :=
proc(n :Dom : :Integer)
  option remember ;
  local i, value, previous, previousprevious ;
begin
  if n=0 then return(0) ; end_if ;
  if n=1 then return(1) ; end_if ;
  previous :=0 ;
  value :=1 ;
  for i from 2 to n do
    operations := operations+1 ;   // Count operations
    previousprevious := previous ;
    previous        := value ;
    value           := previousprevious+previous ;
  end_for ;
  return(value) ;
end_proc :

fibonacci_rec_remember :=
proc(n : Dom : :Integer)
  option remember ;
begin
  if n=0 then return(0) ; end_if ;
  if n=1 then return(1) ; end_if ;
  operations  := operations+1 ;   // Count operations
  return(fibonacci_rec_remember(n-1)+
         fibonacci_rec_remember(n-2)  ) ;
end_proc :
printentry :=
proc(x)
begin
  userinfo(0, NoNL, x) ;
  userinfo(0, NoNL, "\t") ;
end_proc :
```

FIG. 7.1.1. Measurement of the complexity of the different implementations of the Fibonacci sequence

```
for n from 0 to 30 do

  printentry(n);

  operations :=0;
  printentry(fibonacci_rec(n));
  printentry(operations);

  operations :=0;
  fibonacci_array(n);
  printentry(operations);

  operations :=0;
  fibonacci_loop(n);
  printentry(operations);

  operations :=0;
  fibonacci_rec_remember(n);
  printentry(operations);
  // Clear remember table;
  fibonacci_rec_remember :=
    subsop(fibonacci_rec_remember,5=NIL);

  print();
end_for :
```

PROBLEM 7.1.2. Why is Fibonacci rec so slow ? (Trace its execution)

How to avoid this ?

Don't throw away the results !

The `remember` option.

Time consumption :
– Fibonacci rec : exponential
– Fibonacci rec_remember : linear
– Fibonacci array : linear
– Fibonacci loop : linear

Memory consumption :
– Fibonacci rec : exponential
– Fibonacci rec_remember : linear
– Fibonacci array : linear

– Fibonacci loop : constant

Easiness to write :

– Fibonacci rec : straight forward
– Fibonacci rec_remember : straight forward
– Fibonacci array : easy
– Fibonacci loop : harder

7.2. The P=NP conjecture

7.2.1. Case study : Satisfying a well formed formula of propositional logic. We consider a well formed formulas of propositional logic.

Définition. A choice of truth values for A, B, C, ... *satisfies* P if P evaluates to true.

Exemple. A true and B true satisfies $A \wedge B$.

A true and B false do not satisfy $A \wedge B$.

Problem 7.2.1. Given the truth values of A, B, C, ... check if they satisfy P.

Algorithm ?

Complexity ?

Définition. A wff P is *satisfiable* if at least one choice of truth values for A, B, C, ... satisfies P.

I.e. P is not a contradiction.

Exemple. $A \wedge B$ is satisfiable, whereas $A \wedge A'$ is not.

Problem 7.2.2. Testing if a wff P is satisfiable.

Algorithm ?

Complexity ?

Is this the best algorithm ?

Does there exist a polynomial algorithm ?

That's an instance of a problem where :

– checking a solution is easy (polynomial)
– finding the solution seems to be hard (the best known algorithm is exponential)

Exemple. Checking if the solution of a crossword puzzle is correct is easy.

Solving the crossword puzzle is not !

Exemple. Backpack problem with $size = 23$, and $objects = 5, 7, 7, 20, 17, 5, 11, 5, 19, 14$.

7.2.2. Polynomial and Non Deterministic Polynomial problems.

Définition. Problems as above are called *NP* (*Non-deterministic Polynomial*)

P : collection of all polynomial problems

NP : collection of all NP problems

Problem 7.2.3. P = NP ?

In other words :

If it's easy to check a potential solution of a problem,

is there necessarily an easy way to find the solution ?

Intuitively, no. This cannot be true.
Well, despite a lot of research, we still have no proof!
This is the main problem of theoretical computer science since 30 years.
So, this is YOUR job to find the solution!

Troisième partie

Sets and Combinatorics

CHAPITRE 8

Sets (section 3.1)

8.1. Introduction

Problem p 161

Goal :

– Introduce the formalism of set theory
– Use it to solve counting problems

8.2. Sets and basic manipulations of sets

8.2.1. Definition.

DÉFINITION. A *set* A is a collection of objects.

Defining property : you can ask whether an object a belongs to A or not.
Objects or elements : $a, b, c, ...$
Sets : $A, B, C, ...$
Predicates :
– $a \in S$: a belongs to S, or a is an element of S, or S contains a ;
– $a \notin S$: a does not belong to S.

Two sets A and B are *equal* ($A = B$) if they contain the same elements :

$$(\forall x)(x \in A) \Leftrightarrow (x \in B)$$

8.2.2. How to define a set S ? You have to describe which elements are in the set S and which are not.

– List the elements of the set S :

$$S := \{a, b, c\}$$

$a \in S$ is true, whereas $d \in S$ is false.
The order and the repetition are irrelevant :

$$\{a, b, c\} = \{a, b, b, b, c\} = \{c, b, a\}$$

Note that a and $\{a\}$ are two different things. The first is the object a, whereas the second one is the set containing a, and no other elements.

– Provide a predicate which determines which element are in S :

$$S := \{x \,|\, P(x)\}$$

$$S := \{x \,|\, x \text{ is an integer and } 3 < x \leq 7\}$$

$5 \in S$ is true, whereas $3 \notin S$, and no car belongs to A.

– Use a *recursive definition*, as for the set of well formed formulas :
S contains basic propositions $A, B, C, \ldots$
If P and Q belong to S, then P', $P \wedge Q$, $P \vee Q$, $P \to Q$, $P \leftrightarrow Q$ also belong to S.

Some usual sets :

– The empty set :

$$\emptyset := \{x \,|\, false\}$$

– Numbers :

$$\mathbb{N} := \{x \,|\, x \text{ is a non negative integer }\}$$
$$\mathbb{Z} := \{x \,|\, x \text{ is an integer }\}$$
$$\mathbb{Q} := \{x \,|\, x \text{ is a rational number }\}$$
$$\mathbb{R} := \{x \,|\, x \text{ is a real number }\}$$
$$\mathbb{C} := \{x \,|\, x \text{ is a complex number }\}$$
$$S^* := \{x \,|\, x \in S \text{ and } x \neq 0\}$$
$$S^+ := \{x \,|\, x \in S \text{ and } x \geq 0\}$$

8.2.3. Relationships between sets.

– A is a *subset* of B ($A \subseteq B$) iff any element of A is in B :

$$(\forall x)(x \in A) \Rightarrow (x \in B)$$

– A is a *proper subset* of B ($A \subset B$) iff A is a subset of B but they are not equal :

$$(\forall x)(x \in A) \Rightarrow (x \in B) \text{ and } (\exists x)(x \in B) \wedge (x \notin A)$$

EXERCICE 28. [**1**, Exercise 10 p. 117]

$$R := \{1, 3, \pi, 4, 1, 9, 10\},\ S := \{\{1\}, 3, 9, 10\}$$

$$T := \{1, 3, \pi\},\ U := \{\{1, 3, \pi\}, 1\}$$

(1) $S \subseteq R$?
(2) $1 \in R$?
(3) $1 \in S$?
(4) $1 \subseteq U$?
(5) $\{1\} \subseteq T$?
(6) $\{1\} \subseteq S$?
(7) $T \subset R$?
(8) $\{1\} \in S$?
(9) $\emptyset \subseteq S$?
(10) $T \subseteq U$?
(11) $T \in U$?
(12) $T \notin R$?
(13) $T \subseteq R$?
(14) $S \subseteq \{1, 3, 9, 10\}$?

8.2.4. The powerset of a set.

The powerset $\wp(S)$ of a set S is the set of all subsets of S.

EXEMPLE. What are the elements of the following powersets ?

(1) $\wp(\{1,2\}) =$
(2) $\wp(\{1\}) =$
(3) $\wp(\emptyset) =$

EXERCICE 29. What is the size of $\wp(S)$ if S has n elements ?

8.2.5. Binary and unary operations.

DÉFINITION. $\bullet$ is a *unary operation* on a set S if for every element x of S, $\bullet x$ exists, is unique and is a member of S.
$\bullet$ is a *binary operation* on a set S if for every ordered pair (x, y) of elements of S, $x \bullet y$ exists, is unique and is a member of S.

Examples :

(1) is $+$a binary operation on $\mathbb{N}$?
(2) is $+$a binary operation on $\{0, 1, 2\}$?
(3) is $\times$a binary operation on $\{0, 1\}$
(4) is $-$a binary operation on $\mathbb{N}$?
(5) is $/$a binary operation on $\mathbb{Z}$?
(6) are $+$,$-$, and $\times$binary operations on $\mathbb{Q}$?
(7) are $+$,$-$, $\times$and $/$binary operations on $\mathbb{Q}^*$?
(8) is $\sqrt{x}$ a unary operation on $\mathbb{R}^+$?
(9) are $\wedge$, $\vee$, $\rightarrow$, $'$, ... operations on $\{true, false\}$? on wff ?

A binary operation can be defined by its *multiplication* table.

EXEMPLE. $(\{true, false\}, \wedge)$

$\wedge$	$false$	$true$
$false$	$false$	$false$
$true$	$false$	$true$

An operation does not necessarily need to have a particular meaning.

EXEMPLE. $(\{2, 5, 9\}, \bullet)$

$\bullet$	2	5	9
2	2	5	2
5	2	9	5
9	9	2	5

$5 \bullet 9 =?$

8.3. Operations on sets

8.3.1. The universe of discourse. We have seen operations on numbers, or other objects.

We can also define operations on sets (union, intersection, …).

We need a set that contains all the sets we want to deal with.

It would be tempting to consider the set of all sets.

Problem 8.3.1. Let S be the set of all sets that does not contain themselves.

Does S contain itself ?

– if S does not contain itself, then it contains itself !
– if S contains itself, then it does not contain itself !

There is a strange loop here which yields a contradiction : S cannot exist.

For a similar reason, *the set of all sets cannot exist.*

See *Gödel Esher Bach*[**2**] for a lot of similar fun stuff with strange loops.

That's been a major source of trouble and work a century ago, when mathematicians tried to define a strong basis for set theory.

To be safe, we always work inside a set big enough to contain all the sets we need, but small enough to avoid the strange loop above.

Définition. This set is the *universal set*, or the *universe of discourse.*

We define operations on the powerset of the universal set.

8.3.2. Union, intersection, complement and Cartesian product. We define operations on subsets of the universe of discourse.

Définition. The *union* of A and B is the set :

$$A \cup B := \{x \,|\, x \in A \text{ or } x \in B\}$$

Définition. The *intersection* of A and B is the set :

$$A \cap B := \{x \,|\, x \in A \text{ and } x \in B\}$$

Définition. The *complement* of A is the set :

$$A' := \{x \,|\, x \notin A\}$$

Définition. The *set difference* of A and B is the set :

$$A - B := \{x \, (x \in A) \text{ and } (x \notin B)\}$$

Définition. The *Cartesian product* (or *cross product*) of A and B is the set :

$$A \times B := \{(x, y) \,|\, x \in A, y \in B\}$$

Notation 8.3.2. $A^2 = A \times A$ is the set of all ordered pairs of elements of A ;
$A^n = A \times \cdots \times A$ is the set of all n-uples of elements of A.

Définition. *Venn diagrams.*

EXERCICE 30. *Let* $A := \{1, 2\}$ *and* $B := \{2, 3, 4\}$.

(1) $A \cup B =$,
(2) $A \cap B =$,
(3) $A - B =$,
(4) $A' =$ (assuming that the universe of discourse is $\{1, 2, 3, 4\}$),
(5) $A \times B =$,
(6) $A^3 =$.

EXERCICE 31. Let A be a set of size n, and B a set of size m.

(1) what is the size of $A \cup B$:
(2) what is the size of $A \cap B$:
(3) what is the size of A' ?
(4) what is the size of $A \times B$:

8.3.3. Set identities.

PROPOSITION. *Let A and B be to sets. Then,*
$A \cap B = B \cap A$ *(commutative property)*

DÉMONSTRATION. $x \in A \cap B$
$\Leftrightarrow (x \in A) \wedge (x \in B)$ by definition of $\cap$
$\Leftrightarrow (x \in B) \wedge (x \in A)$ by commutativity of $\wedge$
$\Leftrightarrow x \in B \cap A$ by definition of $\cap$ □

The commutativity of $\cap$derives directly from the commutativity of $\wedge$.
All identities on logic operators induce identities on set operators.

PROPOSITION. *Let A, B, and C be two sets. Then,*
$A \cup B = B \cup A$ *(commutative property)*
$A \cap B = B \cap A$ *(commutative property)*
$(A \cup B) \cup C = A \cup (B \cup C) = A \cup B \cup C$ *(associative property)*
$(A \cap B) \cap C = A \cap (B \cap C) = A \cap B \cap C$ *(associative property)*
$A \cup A' = S$ *(complement, S is the universe of discourse)*
$A \cap A' = \emptyset$ *(complement)*
$(A \cup B)' = A' \cap B'$ *(de Morgan's)*
$(A \cap B)' = A' \cup B'$ *(de Morgan's)*
$A - B = A \cap B'$

8.4. Countable and uncountable sets

8.4.1. Enumerations. If a set S has k elements, these can be listed one after the other :

$$s_1, s_2, \ldots, s_k$$

DÉFINITION. $s_1, s_2, \ldots, s_k$ is called an *enumeration* of the set S.

EXEMPLE. $1, 5, 4, 3$, $5, 1, 3, 4$, and $4, 1, 3, 5$ are enumerations of the set $\{1, 5, 4, 3\}$.

When a set S is infinite, you can not enumerate all of it's elements at once. However, some times, you can still pickup a first element s_1, then a second s_2, and so on forever. This process is an enumeration of the elements of S if :
– Any element of S gets enumerated eventually ;
– No element is enumerated twice.

EXEMPLE. $0, 1, 2, 3, 4, \dots$ is an enumeration of the non-negative integers.

8.4.2. Denumerable and Countable sets.

DÉFINITION. A infinite set S is *denumerable* if it has an enumeration

$$s_1, s_2, \dots, s_n, \dots$$

A set S is *countable* if it's either finite or denumerable.

EXEMPLE. Are the following sets countable ?

(1) The set of all students in this class.
(2) The set $\mathbb{N}$ of the non-negative integers (enumeration : $0, 1, 2, 3, 4, \dots$) ;
Actually, the order does not need to be logical.
This is a perfectly legal enumeration : $0, 10, 5, 11, 4, 2, 17, 2, 1, 28, 3$
(3) The set of the even integers (enumeration : ;
(4) The set of the prime integers (enumeration : ;
(5) The set $\mathbb{Z}$ of the integers (enumeration :) ;
(6) The set $\mathbb{N} \times \mathbb{N}$
(e.g. all points in the plane with non-negative integer coordinates) :
(7) Enumeration : $(0, 0), (0, 1), (1, 0), (2, 0), (1, 1), (0, 2), (3, 0), \dots$
(8) The set $\mathbb{Z} \times \mathbb{Z}$ of all points in the plane with integer coordinates :
Enumeration : $(0, 0), (0, 1), (1, 0), (0, -1), (-1, 0), (2, 0), \dots$
(9) The set $\mathbb{Q}$ of rational numbers :

THÉORÈME. *The following sets are countable :*

(1) *Any subset $A \subset B$ of a countable set B.*
(2) *The union $A \cup B$ of two countable sets A and B is countable.*
(3) *The union $A_1 \cup A_2 \cup \dots \cup A_i$ of a finite number i of countable sets $A_1, \dots, A_i$.*
(4) *The union $A_1 \cup A_2 \cup \dots \cup A_i \cup \dots$ of a countable number of countable sets.*
(5) *The cross product $A \times B$ of two countable sets A and B is countable.*
(6) *The cross product $A_1 \times \dots \times A_i$ of a finite number of countable sets $A_1, \dots, A_i$.*

DÉMONSTRATION. Enumerations of those sets can be constructed in the following way : □

(1) Enumerate the elements of B, and ignore those that are not in A.
(2) Enumerate the elements of A and B alternatively $(a_1, b_1, a_2, b_2, \dots)$.

(3) Induction, using 2.

(4) As for $\mathbb{N} \times \mathbb{N}$: $a_{1,1}, a_{2,1}, a_{1,2}, a_{3,1}, a_{2,2}, a_{1,3}, \ldots$ ($a_{i,j}$ is the jth element of A_i).

(5) As for $\mathbb{N} \times \mathbb{N}$: $(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_1, b_3), (a_2, b_2), (a_3, b_1), \ldots$.

(6) Induction, using 5.

8.4.3. Uncountable sets ?

Remarque. All the sets we have seen so far are countable.

Problem 8.4.1. Are there any uncountable sets ?

Théorème. *(Cantor)* $\mathbb{R}$ *is uncountable.*

Lemme. *Any real number* $x \in [0,1)$ *can be written uniquely in decimal form :*

$$x = 0.d_1 d_2 d_3 d_4 \cdots d_i \cdots ,$$

where the d_i *are digits between* 0 *and* 9*, and the sequence does not end by an infinite number of* 9.

Exemple. Here is how some classical numbers can be written :
- $0 = 0.0000\cdots$
- $0.5 = 0.50000\cdots$
- $0.5 = 0.49999\cdots$ (Same number as above, we don't use this form)
- $\frac{1}{3} = 0.333333\cdots$
- $\pi = 3.141592653\cdots$

Let's jump to the proof of the theorem, without detailing the proof of this lemma.

Démonstration. (Of the theorem) : *Cantor diagonalization method.*
It's sufficient to prove that $[0,1)$ is not countable.
Let's do this by contradiction : assume $[0,1)$ is countable.
Let $s_1, s_2, \ldots, s_i, \ldots$ be an enumeration of $[0,1)$.
Goal : construct an element which is not in this enumeration.
Using the lemma, we can write the s_i as follow :

$$\begin{array}{ccccccccc}
s_1 & = & 0, & \underline{d_{1,1}} & d_{1,2} & d_{1,3} & d_{1,4} & d_{1,5} & \cdots \\
s_2 & = & 0, & d_{2,1} & \underline{d_{2,2}} & d_{2,3} & d_{2,4} & d_{2,5} & \cdots \\
s_3 & = & 0, & d_{3,1} & d_{3,2} & \underline{d_{3,3}} & d_{3,4} & d_{3,5} & \cdots \\
s_4 & = & 0, & d_{4,1} & d_{4,2} & d_{4,3} & \underline{d_{4,4}} & d_{4,5} & \cdots \\
s_5 & = & 0, & d_{5,1} & d_{5,2} & d_{5,3} & d_{5,4} & \underline{d_{5,5}} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

We want to construct an element x which is not in this enumeration.
Let's look at an example :

$$\begin{array}{ccccccccc}
s_1 & = & 0, & \underline{1} & 5 & 7 & 3 & 5\cdots \\
s_2 & = & 0, & 0 & \underline{0} & 4 & 7 & 3\cdots \\
s_3 & = & 0, & 4 & 5 & \underline{7} & 0 & 3\cdots \\
s_4 & = & 0, & 9 & 7 & 3 & \underline{5} & 7\cdots \\
s_5 & = & 0, & 4 & 3 & 8 & 1 & \underline{0}\cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\ddots \\
 & & & & & & & \\
x & = & 0, & 0 & 1 & 0 & 0 & 1\cdots
\end{array}$$

The trick : let $x_i = 1$ if $d_{i,i} = 0$ and $x_i = 1$ else.
Then, $x \neq s_1$. But also, $x \neq s_2$, $x \neq s_3$, ...
Therefore x is indeed never enumerated ! Contradiction !
Conclusion : $[0, 1)$ is not countable. □

Remarque. We can deduce that many sets like $\mathbb{C}$, or $[5.3, 10)$ are uncountable.
A similar proof can be used to prove that many other sets are uncountable.
This includes the set of all infinite strings, or the set $\wp(\mathbb{N})$ of all subsets of $\mathbb{N}$.

Résumé 8.4.2. We have a hierarchy of bigger and bigger sets :

(1) The empty set
(2) Finite sets (the empty set, sets with 1 element, sets with 2 elements, ...)
(3) Denumerable sets : $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, ...
(4) Uncountable sets : $\mathbb{R}$, $\mathbb{C}$, ...
(5) ...

Problem 8.4.3. Is there any set bigger than $\mathbb{N}$ and smaller that $\mathbb{R}$? Nobody knows !

8.5. Conclusion

We are now done with basic properties of sets. We have seen enough formalism to go on to the next section, *Combinatorics* :

How to count the objects in a finite set ?

CHAPITRE 9

Combinatorics (section 3.2...3.5)

9.1. Introduction

Goal : answer *"How many ?"* questions :
– How many ways to climb a stair ?
– How many rows in a truth table ?
– How many trees ?

Applications :
– Analysis of algorithms
– Statistics
– Size of a database

9.2. Counting

9.2.1. Preliminaries.

Définition. The *size* of a finite set is the number of elements in this set.
The size of S is denoted by $|S|$ (or, often in the literature, $\#S$).

Note. In the sequel, we only consider finite sets !

A*"how many ?"* question is usually formalized as follow :

(1) Define the set S of the objects to be counted
(2) What is the size of S ?

Techniques to answer this question :
– breakup S as some combination of simpler sets
– transform S into a set, the size of which is already known

9.2.2. The addition principle.

Exemple. A small company wants to know how many computers it owns, given that it owns 5 Mac's, and 3 UNIX stations.
Let :
– C be the sets of computers,
– M be the set of Mac's ($|M| = 5$),
– U be the set of UNIX stations ($|U| = 3$).
Then, $C = M \cup U$.
What is the size of C ?

$$|C| = |M| + |U| = 5 + 3 = 8$$

THÉORÈME. *(Addition principle) Let A and B be disjoint sets (e.g. $A \cap B = \emptyset$). Then :*

$$|A \cup B| = |A| + |B| .$$

EXEMPLE. Imagine 2 of the 5 Mac's are actually running Linux.
What is the size of C ?
Well, $|C| = 6 < 8$.

So, the disjoint hypothesis is important !
What to do when the sets are not disjoint ?

THÉORÈME. *Let A and B be two sets. Then :*

$$|A \cup B| = |A| + |B| - |A \cap B| .$$

DÉMONSTRATION. We just have to split A, B, and $A \cup B$ into unions of disjoint subsets :

$$A = (A - B) \cup (A \cap B)$$

$$B = (B - A) \cup (A \cap B)$$

$$A \cup B = (A - B) \cup (A \cap B) \cup (B - A).$$

(Practice : draw and prove those set identities !).
Then,

$$\begin{aligned} |A| + |B| - |A \cap B| &= (|A - B| + |A \cap B|) + (|B - A| + |A \cap B|) - |A \cap B| \\ &= |A - B| + |A \cap B| + |B - A| \\ &= |A \cup B| \end{aligned}$$

□

EXEMPLE. In the example above, $|C| = |M| + |U| - |M \cap U| = 5 + 3 - 2 = 6$.

EXERCICE 32. There are 24 students in a class, all of them are math majors or computer science major. 10 are math majors, and 20 are computer science majors. How many double-major are there in this class ?

9.2.3. The multiplication principle.

EXEMPLE. You are a car dealer. For each car you sell, you propose several options (color, 2WD/4WD, airbag, ABS, radio, ...). Of course, you would like to always have all possible combinations of options in stock. On the other hand, you cannot afford to store too many cars in your parking lot. So you need to know how many possible combinations of options there are.
– 2WD / 4WD
– Black / Yellow / Green

How many combinations are there ?

First solution : draw a *decision tree.*

The choices are independent !

The number of sub-branches at each level is constant.

Therefore, we have $2 \cdot 3 = 6$ choices.

Let's formalize this.

Let C be the set of all choices. Each choice can be represented by a couple.

For example a yellow 2WD car with black seats can be represented as :

$$(2, Y)$$

So, C can be viewed as the cross product :

$$\{2, 4\} \times \{B, Y, G\}.$$

THÉORÈME. *Let A and B be two sets. Then,*

$$|A \times B| = |A| \cdot |B|.$$

EXEMPLE. In the example above, $|C| = |\{2,4\} \times \{B, Y, G\}| = |\{2,4\}| \cdot |\{B, Y, G\}| = 2 \cdot 3 = 6$.

EXERCICE 33. How many possible different phone numbers are there ?

9.2.3.1. *Strings.* Strings are commonly used objects in computer science. "how many" questions often be translate into questions about sets of strings.

DÉFINITION. Let S be a set, that we call *alphabet.*

Elements of S are called *letters.*

A *string* over S is a sequence $(s_1, s_2, \dots, s_n)$ of elements of S.

$\lambda := ()$ is the empty string.

The set of all strings of length n is $S^n = S \times \cdots \times S$.

The set of all finite strings over S is denoted S^*.

A *language* is a set of strings.

EXEMPLE. Let $S := \{0, 1\}$. A string over S, like `001001` is called a binary string.

Let $S := \{a, b, \dots, z\}$. Then `bonjour` is a string over S of length 7.

A phone number `3032733462` is a string over $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

The set of all English words is a language.

EXEMPLE. Count the following :

(1) binary strings of length 4 ?

(2) strings of length n over an alphabet containing p letters ?

9.2.4. Decision trees.

EXEMPLE. A kid is allowed to pick 3 candies (one after another), out of one jelly bean, one lollipop and 2 gummy bears (one blue, one yellow). In how many ways can he do this ?

Draw a decision tree.

The choices are not independent

The multiplication principle does not apply.

However, the number of choices at each step does not change !

So the answer is : $5 \cdot 4 \cdot 3 = 60$.

EXERCICE 34. Count the following :

(1) Strings of length 4 over $\{1, 2, 3, 4, 5\}$ with no repetitions.
(2) Strings of length n over $\{1, \dots, n\}$ with no repetitions.

EXERCICE 35. How many ways to toss a coin five times without two heads in a row ?

Note : the decision tree is irregular.

9.2.5. Combining those principles together.

EXEMPLE. Count the following :

Exercise 18-19 p. 195
– Strings of length n over $\{1, \dots, n\}$ with repetitions.
– Strings of length n over $\{0, 1\}$ with two consecutive 1.
– How many four-digits numbers begin with 4 or 5 ?
– How many ways to toss a coin seven times with two heads in a row ?

9.2.6. Principle of Inclusion and Exclusion.

EXEMPLE. All the guests at a dinner party drink coffee, chocolate or tea ; 11drink coffee, 9 drink tea, 10 drink chocolate ; 3 drink coffee and tea, 5 drink tea and chocolate ; 6 drink coffee and chocolate

Formalization : what is the size of $|A \cup B \cup C|$?

Generalization ?

THÉORÈME. *(Principle of Inclusion and exclusion)*
Let $A_1, \dots, A_n$ be n sets. Then,

$$\begin{aligned}
|A_1 \cup \cdots \cup A_n| &= |A_1| + \cdots + |A_n| \\
&\quad - |A_1 \cap A_2| - \cdots - |A_{n-1} \cap A_n| \\
&\quad + |A_1 \cap A_2 \cap A_3| + \cdots |A_{n-2} \cap A_{n-1} \cap A_n| \\
&\quad + (-1)^{n+1} |A_1 \cap \cdots \cap A_n| \\
&= \sum_{I \subset \{1, \cdots, n\}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|
\end{aligned}$$

DÉMONSTRATION. Induction. □

9.2.7. Pigeon Hole principle.

EXERCICE 36. It's early in the morning, and you don't want to wake up your significant whatever by turning on the light. You know you only have black and white socks in your drawer. Is it possible to exit the room with at least two socks with the same color ?

THÉORÈME. *(Pigeon hole principle)*

If more than k items are placed in k bins, then at least one bin has more than one item inside.

EXEMPLE. How many people do you need to have in a room to ensure at least two of them have the same birthday ?

9.2.8. Permutations : sequences without repetitions.

EXEMPLE. How many ways to choose a committee with 1 president and 1 vice-president out of a group of n persons.

This amount to find the number of strings of length 2 over a set of size n, without repetitions. This is $n(n-1)$.

DÉFINITION. We denote by $P(n,k)$ the number of strings of length k over a set of size n without repetitions.

EXEMPLE. $P(n,0) =\ $, $P(n,1) =\ $, $P(n,n) =\ $, $P(1,2) =\ $.

THÉORÈME. $P(n,k) = n(n-1)\cdots(n-k+1) = \frac{n!}{(n-k)!}$.

9.2.9. Combinations.

EXEMPLE. How many ways to choose 3 voluntaries out of a group of 5 persons ?

DÉFINITION. We denote by $C(n,k)$ the number of subsets of size k of a set of size n.

EXEMPLE. $C(n,0) =\ $, $C(n,1) =\ $, $C(n,n) =\ $, $C(1,2) =\ $.

THÉORÈME. $C(n,k) = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} = \frac{n!}{k!(n-k)!}$.

EXEMPLE. Ex. 51 p. 210 :

A committee of 8 students is to be selected from a class consisting of 19 freshmen and 34 sophomores.

(1) In how many ways can 3 freshmen and 5 sophomores be selected ?
(2) In how many ways can a committee with exactly 1 freshman be selected ?
(3) In how many ways can a committee with at most 1 freshman be selected ?
(4) In how many ways can a committee with at least 1 freshman be selected ?

EXEMPLE. What is the total number of subsets of S ?

EXEMPLE. What is the number of paths between Stratton Hall and Golden Tourism Information center ?

9.2.10. Combinations with repetitions.

Exemple. In how many ways can you distribute 10 lollipops between 4 kids.

Théorème. *The number of combinations with repetitions of k elements of a set S of size n is $C(n+k-1,n)$.*

Exemple. In how many ways can you distribute 10 lollipops between 4 kids, so that every kid get at least one lollipop ?

9.2.11. Summary.

Addition principle (A and B disjoint)	$\lvert A\cup B\rvert = \lvert A\rvert + \lvert B\rvert$
Principle of Inclusion-Exclusion	$\lvert A\cup B\rvert = \lvert A\rvert + \lvert B\rvert - \lvert A\cap B\rvert \lvert A_1\cup\cdots\cup A_n\rvert = \cdots$
Multiplication principle	$\lvert A\times B\rvert = \lvert A\rvert\cdot\lvert B\rvert \lvert A^k\rvert = \lvert A\rvert^k$
Sequences of k elements of S ($\lvert S\rvert = n$)	n^k
Permutations :	$P(n,k) = \frac{n!}{(n-k)!}$
Sequences of k elements of S, w/o repetitions	
Sequences of k elements of $\{0,1\}$, w/o successive 1	Fibonacci : $F(k)$
Combinations : subsets of size k of S	$C(n,k) = \frac{n!}{k!(n-k)!}$
All subsets of S	2^n
Combinations with repetitions	$C(n+k-1,k)$

9.3. Properties of binomial coefficients ; Combinatorial proofs

Exemple. How many subsets of size 14 of a set of size 15 ?

Théorème. $C(n,k) = C(n,n-k)$

Démonstration. □

(1) Algebraic proof
(2) Combinatorial proof

Définition. *Combinatorial proof* :
Goal : prove that two quantities a and b are equal
Technique :

(1) construct a set A of size a ;
(2) construct a set B of size b ;
(3) construct a *bijection* between A and B

Problem 9.3.1. Recursive computation of $C(n,k)$?

Théorème. $C(n,k) = C(n-1,k) + C(n-1,k-1)$

Démonstration. □

(1) Algebraic proof
(2) Combinatorial proof

EXEMPLE. Compute $C(7,4)$, without calculator.

DÉFINITION. Pascal triangle :

$n\backslash k$	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

THÉORÈME. $C(n,0) + \cdots + C(n,n) = 2^n$

DÉMONSTRATION. □

(1) Algebraic proof ?
(2) Combinatorial proof

9.3.1. The binomial theorem.

PROBLEM 9.3.2. Compute $(x+y)^n$.

THÉORÈME. *There is a formula for expanding* $(x+y)^n$ *:*

$$\begin{aligned}(x+y)^n &= \sum_{k=0}^{k=n} C(n,k)x^{n-k}y^k \\ &= C(n,0)x^n + C(n,1)x^{n-1}y + C(n,2)x^{n-2}y^2 + \cdots + C(n,n)y^n\end{aligned}$$

9.4. Conclusion

Quatrième partie

Relations Functions and Matrices

CHAPITRE 10

Relations (section 4.1, 4.2 and 4.4)

10.1. Introduction

A mere set of words would not make a good dictionary.
It would be a pain to find a particular word !
A usable dictionary has some structure : the words are sorted.
In general, the more *structure* a set have, the more useful it is.
A way to bring structure into this set is to describe the *relations* between its elements, or between its elements and the elements of another set.
In this section we will see how we can formalize and study relations.

EXEMPLE. Imagine you want to build a house.
Figure 10.1.1 shows the tasks that need to be completed.
Let $S := \{F, W, E, I, O, R\}$ be the set of all tasks.
Problem : can we do the tasks in any order ?
For example, it would be better to build the walls AFTER the foundations !
S in itself does not contain enough information to choose a correct order.
Set of constraints : $\rho := \{(F, W), (W, O), (W, E), (R, E), (R, I)\}$.
This set of constraints gives some structure to S, and makes it useful.

10.2. Relations

10.2.1. Definitions.

DÉFINITION. A *binary relation* on a set S is a subset ρ of $S \times S$.
Let x and y be two elements of S.
Then x is *in relation* with y (denoted $x\ \rho\ y$) iff $(x, y) \in \rho$.

EXEMPLE. Let ρ be the relation “is a prerequisite for” :

$$\rho := \{(F, W), (W, O), (W, E), (R, E), (R, I)\}$$

Then, $(F, W) \in \rho$, but $(I, O) \notin \rho$.

FIG. 10.1.1. Tasks to build a house

FIG. 10.1.2. Constraints between the tasks to build a house

So, $F\ \rho\ W$ is true, whereas $I\ \rho\ O$ is false.

A relation can be defined by a property.

EXERCICE 37. Let $S := \{1,2,3,4,5\}$. Draw the relations defined by :

(1) $x\ \rho_1\ y$ iff $x = y$;
(2) $x\ \rho_2\ y$ iff $x \leq y$;
(3) $x\ \rho_3\ y$ iff x divides y ;
(4) $x\ \rho_4\ y$ iff $x - y$ is even.

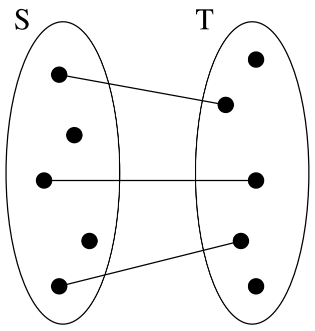
DÉFINITION. A *binary relation* between two sets S and T is a subset ρ of $S \times T$.
A *n-ary relation* between n sets $S_1, \ldots, S_n$ is a subset ρ of $S_1 \times \cdots \times S_n$.

EXEMPLE. Let M be a set of male and F a set of female students.
We can define the relation "is married to".

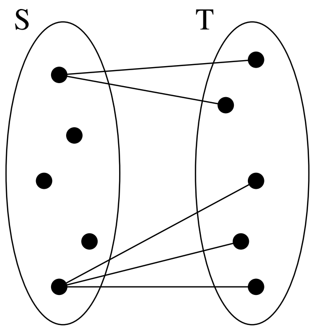
10.2.2. Basic properties of relations.

Is there anything particular about the relation "is married to" ?

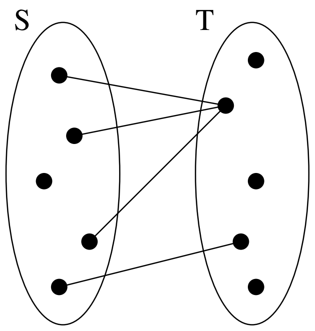
DÉFINITION. Let ρ be a relation between S and T.
– S is *one-to-one* if any element of S and T appears at most once in ρ ;



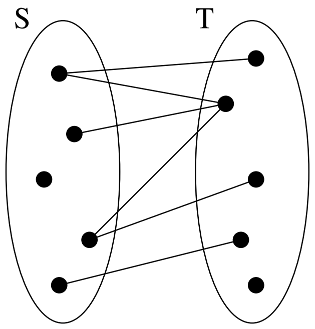
– S is *one-to-many* if any element of T appears at most once in ρ ;



– S is *many-to-one* if any element of S appears at most once in ρ ;



– S is *many-to-many* in all other cases.



Exemple. Is x lower or equal to itself ?

Définition. A relation ρ on a set S is *reflexive* iff

$$(\forall x \in S)\ x\ \rho\ x.$$

Exemple. Assume x is equal to y. Is y equal to x ?

Définition. A relation ρ on a set S is *symmetric* iff

$$(\forall x \in S)(\forall y \in S)\ (x\ \rho\ y) \leftrightarrow (y\ \rho\ x).$$

Exemple. Assume $x \leq y$ and $y \leq x$. What can you say about x and y ?

Définition. A relation ρ on a set S is *antisymmetric* iff

$$(\forall x \in S)(\forall y \in S)[(x\ \rho\ y) \text{ and } (y\ \rho\ x)] \rightarrow (x = y).$$

Exemple. Assume $x < y$ and $y < z$. What can you say about x and z ?

Définition. A relation ρ on a set S is *transitive* iff

$$(\forall x \in S)(\forall y \in S)(\forall z \in S)[(x\ \rho\ y) \text{ and } (y\ \rho\ z)] \rightarrow (x\ \rho\ z).$$

Exemple. What are the properties of the following relations

(1) $<, \leq, =$;
(2) “is married to” ;
(3) “is a friend of” ;
(4) “has the same color than”.

FIG. 10.2.1. Constraints between the tasks to build a house

10.2.3. Operations on relations.

A relation is basically a set. So, we can use all usual set operations on relations.

REMARQUE. Let S and T be two sets.
The powerset $\wp(S \times T)$ is the set of all binary relations between S and T.

DÉFINITION. Let ρ and σ be two relations between S and T.
We can construct the following new relations :
– union : $\rho \cup \sigma$,
– intersection : $\rho \cap \sigma$,
– complement : ρ',
– ...

EXEMPLE. Let $S := \mathbb{N}$.

(1) What is the union of $<$and $=$?
(2) What is the intersection of $<$ and $>$?
(3) What is the union of $<$ and $>$?
(4) Is $<$a sub relation of $\leq$?

10.2.4. Closure of a relation.

EXEMPLE. Lets come back to the example of the house building.
Do you have to do the electricity after the foundations ?
Foundations is not a direct prerequisite for electricity.
However, it still needs to be done before electricity.
The relation “has to be done before” is transitive, and contains the relation “is a prerequisite for”. It's the *transitive closure* of “is a prerequisite for”.

DÉFINITION. Let ρ be a relation.
The *transitive closure* of ρ is the smallest transitive relation ρ^*containing ρ.

EXERCICE 38. Draw the transitive closure of “is a prerequisite for”.

PROBLEM 10.2.1. Is there an algorithm for computing the transivite relation of a relation ?

ALGORITHME 10.2.2. *Construction of the transitive closure ρ^*.*

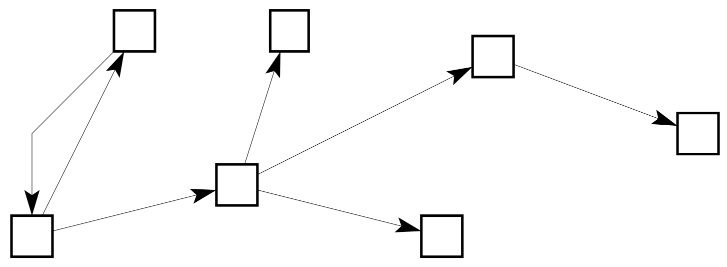
```
(1) $\rho^* := \rho$ ;
(2) Find $x$, $y$, $z$ such that $(x\ \rho^* y)$, $(y\ \rho^* z)$, and $(x\ \not\rho^* z)$ ;
(3) If no such triple exists, $\rho^*$is transitive ; Exit ;
(4) $\rho^* := \rho^* \cup \{(x, z)\}$ ;
(5) Repeat at step 2.
```

REMARQUE. The order in which you add the pairs to ρ is irrelevant.

EXEMPLE. Draw the relation “has to be done before” (Figure 10.4.1).

DÉFINITION. Let ρ be a relation.
The *reflexive closure* of ρ is the smallest reflexive relation containing ρ.
The *symmetric closure* of ρ is the smallest symmetric relation containing ρ.

EXERCICE 39. Consider the following relation :



(1) Draw its reflexive closure
(2) Draw its symmetric closure
(3) Draw its transitive closure
(4) Draw its antisymmetric closure

REMARQUE. The antisymmetric closure of a relation cannot be uniquely defined !

10.3. Equivalence relations

EXEMPLE. Consider a set of cars, and the relation "has the same color as".
What are the properties of this relation ?

DÉFINITION. An *equivalence relation* is a relation which is

(1) Reflexive
(2) Symmetric
(3) Transitive

DÉFINITION. Let ρ be a relation on a set S, and x be an element of S.
The *equivalence class* of x is the set $[x]$of all elements y such that $x \ \rho \ y$.

EXEMPLE. The equivalence class of a car is the set of all cars having same color.

REMARQUE. If $x \ \rho \ y$, then $[x] = [y]$.

EXEMPLE. Consider the relation $\equiv_3$ on $\mathbb{N}$ defined by $x \equiv_3 y$ iff 3 divides $x - y$.
(we also say that x and y are *congruent* modulo 3 : $x \equiv y \ (mod \ 3)$)
What are the equivalence classes ?

DÉFINITION. A partition of a set S is a collection of subsets $\{A_1, \dots, A_k\}$ of S such that :

(1) $A_1 \cup \dots \cup A_k = S$
(2) $A_i \cap A_j = \emptyset$ for any i, j.

Exemple. Consider a set S of car, whose color are either red, blue, or green.
Define the following subsets of S :
R (red cars), B (blue cars), and G (green cars) .
Then, $\{R, B, G\}$ is a partition of S.

The equivalence relation "has the same color" and the partition of the cars by color are closely related.

Théorème. *In general :*

(1) *An equivalence relation on S defines a unique partition of S.*
(2) *A partition of S defines a unique equivalence relation on S.*

Démonstration. Left as exercise :

(1) Construct the collection $\{A_1, \dots, A_k\}$, and prove it is indeed a partition of S.
(2) Construct the relation ρ, and prove it's indeed an equivalence relation.

$\square$

Définition. Let S be a set, and ρ be a relation on S.
The set of all the equivalence classes is called the *quotient* of S by ρ.

Remarque. This method is used a lot in set theory :
Construction of $\mathbb{Z}$ from $\mathbb{N}$.
Construction of $\mathbb{Z}/n\mathbb{Z}$ (integers modulo n) from $\mathbb{Z}$.
Construction of $\mathbb{Q}$ from $\mathbb{Z}$.

10.4. Partially Ordered Sets

Exemple. What are the properties of the relation "has to be done before" on the tasks to build a house ?

Définition. A *partially ordered set* (or *poset*) is a set S with a relation ρ which is

(1) Reflexive
(2) Antisymmetric
(3) Transitive

Exemple. Which of the following are posets ?

(1) $(\{1, 2, 3, 4\}, \leq)$;
(2) $(\{1, 2, 3, 4\}, <)$;
(3) $(\{1, 2, 3, 4\}, =)$;
(4) $(\{1, 2, 3, 4\}, \rho)$, with $x \ \rho \ y$ iff x divides y.
(5) $(\wp(\{1, 2, 3, 4\}), \subseteq)$.

FIG. 10.4.1. Constraints between the tasks to build a house

10.4.1. Drawing posets ; Hasse diagrams.

DÉFINITION. Some names :
– *Node* (or *vertex*)
– *Predecessor*
– *Immediate predecessor*
– *Maximal element*
– *Minimal element*
– *Least element*
– *Greatest element*

DÉFINITION. The *Hasse diagram* of a poset is a drawing of this poset such that :
– If $x \rho y$, then y is higher than x ;
– The loops are not drawn ;
– There is a segment linking x and y iff x is an immediate predecessor of y.

EXEMPLE. Draw all posets on 1, 2, 3, 4 indistinct nodes.
How many such posets are there on 10 nodes ?

10.4.2. Scheduling. Scheduling a set of tasks consist in allocating tasks to resources, subject to certain constraints.

EXEMPLE. Consider the house building problem, with the following assumptions :
– All tasks take the one day to complete ;
– One worker can only work on one task every day ;
– It does not save time to have two workers working on the same task.
Schedule the tasks between 2 workers to optimize the completion time :

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Worker 1							
Worker 2							

What if there is one worker ?

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Worker 1							

What if there are three workers ?

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day7
Worker 1							
Worker 2							
Worker 3							

PROPOSITION. *Consider the problem of scheduling n tasks on p processors, subject to precedence constraints, with the same assumptions and goal as above.*

- 1 or 2 processors : algorithms in $O(n^2)$;
- 3 processors : complexity unknown ;
- p processors : NP-complete
  (basically the only known algorithm is exhaustive search) ;
- ∞ processors : $O(n)$.

This is typical from scheduling problems, where a very small change in the constraints can make huge differences in the complexity of the problems.

Also, usually the complexity is as follow :

- No constraints : low complexity ;
- Some constraints : higher and higher complexity ;
- More constraints : NP-complete. The only known algorithm is exhaustive search through all plausible cases (*branch-and-cut*) ;
- Even more constraints : still NP-complete, but easier. Indeed more and more cases can be thrown away very early.

Having very good scheduling algorithms is vital in industry, because a 1% difference in completion time (for example) can save thousands of dollars. Consequently, research in this topic is well financed !

10.5. Conclusion

- The more structure a set has, the more useful it is.
- Defining a relation is a way to add structure to a set.
- Relations, posets, equivalence relations, ... are just abstract models of natural notions commonly used in real life.

CHAPITRE 11

Functions (section 4.4)

11.1. Introduction

The notion of function is pretty natural, and used in many domains (programming languages, calculus, ...). So far, the intuitive notion of function was sufficient, but we now need a more formal definition, and we will use relations for this.

We will also see how certain functions, called bijections, provide powerful counting methods.

Exemple. Let S be a set of cars and T be the set of all colors.

Consider the relation “is of color”.

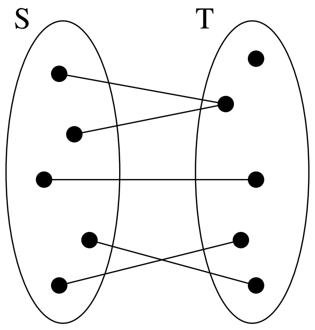
What properties does this relation have ?"

You can speak about THE color of the car : uniquely defined.

11.2. Formal definition of functions

11.2.1. Functions with one variable.

Définition. A *function* $f: S \to T$ between two sets S and T is a relation such that any element s of S is in relation with a unique element of T.



This unique element, denoted $f(s)$ is the *image* of s.

If $f(s) = y$, then s is a *preimage* of y.

S is the *domain* of f ;

T is the *codomain* of f ;

The set $\{f(s)\ ,\ s \in S\}$ is the *range* of f.

Exemple. The identity on a set S if the function $id_S : S \to S$ defined by $id_S(x) := x$.

One way to define a function is to provide an equation, or any other mean that allows to compute the image of an element of the domain.

EXEMPLE. $f(n) := n + 2$;
$f(L) := sort(L)$;
$f(x) := [x]$.

DÉFINITION. *Graph* of a function from $\mathbb{R}$ to $\mathbb{R}$ (or any subset of them) :

EXERCICE 40. Draw the graphs of the functions $f(x) = x^2$, $g(x) = \sqrt{x}$, and $h(x) := |x|$.

11.2.2. Functions with several variables.

The argument of a function does not need to be a simple number.

EXEMPLE. $f(x, y) = x^2 + 2xy$

DÉFINITION. A function with n variables is a function whose argument is a n-uple :

$$f \;:\; S_1 \times \cdots \times S_n \;\rightarrow\; T.$$

EXEMPLE. Here are a few functions with several variables :

– $f \;: \{students\} \times \{test1, test2, final\} \;\rightarrow\; \{1, 2, \ldots, 100\}$

$$f(Jason, test2) := \cdots$$

– $f : \; \{True, False\}^3 \;\rightarrow\; \{True, False\}$

$$f(A, B, C) = (A \wedge B)' \wedge C$$

11.2.3. Equality of functions.

EXEMPLE. Are the functions $f(x) := x$ and $g(x) := |x|$ equal ?

DÉFINITION. Two functions f and g are *equal* iff they have same domain S, same codomain T, and $f(x) = g(x)$ for all x in S.

11.3. Properties of functions

11.3.1. Injective, surjective and bijective functions.

EXEMPLE. Let P be the set of all residents in America ;
let N be the set of all assigned Social Security Number (SSN) ;
let $SSN : \; P \rightarrow N$ be the function which assigns to each person x its SSN $SSN(x)$.

A Social Security Number is useful because it uniquely describes a person !
That is, given an assigned SSN, you can find the person having this SSN.

DÉFINITION. The function SSN is *invertible*.
The *inverse* of the function SSN is the function $SSN^{-1} : \; N \rightarrow P$ such that :
$SSN^{-1}(n)$ is the person having n as SSN.

What properties does a function need to be invertible ?
To be invertible, a function needs to have the two following properties :

DÉFINITION. A function $f : S \rightarrow T$ is *injective* (*one-to-one*) iff

$$(\forall x \in S)(\forall y \in S)\ (f(x) = f(y)) \rightarrow (x = y)$$

"If x and y have the same SSN, then x is the same person as y"

DÉFINITION. A function $f : S \rightarrow T$ is *surjective* (*onto*) iff

$$(\forall y \in T)(\exists x \in S)\ f(x) = y$$

"For any assigned SSN, there is a person which has this SSN."

DÉFINITION. A function $f : S \rightarrow T$ is *bijective* iff it's injective and surjective.

EXERCICE 41. Which one of the following functions from $\mathbb{R}$ to $\mathbb{R}$ are bijective ?

(1) $f(x) := x^2$;
(2) $f(x) := x^3$;
(3) $f(x) := \frac{1}{1+x^2}$;
(4) $f(x) := \exp(x)$;
(5) $f(x) := \log(x)$;

11.3.2. Composition of functions.

DÉFINITION. Let $f : S \rightarrow T$ and $g : T \rightarrow U$ be two functions.
The *composition function* $g \bullet f$ is the function from S to U defined by

$$g \bullet f(x) := g(f(x)).$$

REMARQUE. The standard symbol for composition is an empty circle ! In those notes a full circle is used instead due to a bug in lyx ...

EXERCICE 42. Define $f : \mathbb{R} \rightarrow \mathbb{R}$ by $f(x) := x + 1$, and $g : \mathbb{R} \rightarrow \mathbb{R}$ by $g(x) := x^2$.
What is the value of $g \bullet f(1)$? What is $g \bullet f$?
What is the value of $f \bullet g(1)$? What is $f \bullet g$?

THÉORÈME. *Let $f : S \rightarrow T$ and $g : T \rightarrow U$ be two functions.*

(1) If f and g are injective, then $g \bullet f$ is injective.
(2) If f and g are surjective, then $g \bullet f$ is surjective.
(3) If f and g are bijective, then $g \bullet f$ is bijective.

DÉMONSTRATION. 3. is a direct consequence of 1 and 2.

(1) Assume f and g are injective. We have :

$$(\forall x)(\forall y)\ (f(x) = f(y)) \ \rightarrow\ (x = y)$$
$$(\forall x)(\forall y)\ (g(x) = g(y)) \ \rightarrow\ (x = y)$$

We want to prove that :

$$(\forall x)(\forall y)\ (g \bullet f(x) = g \bullet f(y)) \ \rightarrow\ (x = y)$$

Take x and y such that $g \bullet f(x) = g \bullet f(y)$
By definition $g(f(x)) = g(f(y))$.
Then, since g is injective, $f(x) = f(y)$.
Since f is also injective $x = y$.
We are done !

(2) Left as exercise.

□

11.3.3. Inverse of function.

Exemple. If you take the SSN of a person, and then lookup the person having this SSN, you will get back the original person. That is, composing the function SSN and the function SSN^{-1} yields the identity.

Définition. Let $f : S \to T$ be a function.
If there exists a function $g : T \to S$ such that $g \bullet f = id_S$ and $f \bullet g = id_T$, then g is called the *inverse function* of f, and is denoted f^{-1}.

Théorème. *A function f is bijective iff its inverse f^{-1} exists.*

Exercice 43. Give the inverses (if they exists!) of the following functions :

(1) $f(x) := x - 1$;
(2) $f(x) := x^2$;
(3) $f(x) := x^3$;
(4) $f(x) := \exp(x)$.

11.4. Functions and counting

11.4.1. Injections, surjections, bijections and cardinality of sets.

Exemple. You distribute n different cakes between k child.
Such a distribution can be formalized by a function f :
$f(4) = 5$ means that the 4th cake is given to the 5th child.

(1) Suppose f is injective.
  – What does it mean ?
  – What can you say about n and k ?$k \geq n$
(2) Suppose f is surjective.
  – What does it mean ?
  – What can you say about n and k ?
(3) Suppose f is bijective.
  – What does it mean ?
  – What can you say about n and k ?

Théorème. *Let S and T be two sets.*
If there exists an injective function between S and T, then $|S| \leq |T|$.
If there exists a surjective function between S and T, then $|S| \geq |T|$.
If there exists a bijective function between S and T, then $|S| = |T|$.

Note that we have never given a formal definition of the size of a set.
We just relied on the intuitive notion.
In set theory, the theorem above is actually the definition of cardinality :
– Two sets have the same cardinality (size), iff they are in bijection.
– A set is of size n iff it's in bijection with $\{1, \ldots, n\}$.

11.4.2. Examples of bijections.

11.4.2.1. *Finite and countable sets.* Do you remember how we proved that Z was countable ?

– A set S is finite if there is an enumeration $s_1, \dots, s_k$ of S.
  This enumeration is actually a bijection from $\{1, \dots, k\}$ to S !
– A set S is denumerable if there is an enumeration $s_1, \dots, s_k, \dots$ of S.
  This enumeration is actually a bijection from $\mathbb{N}$ to S !

11.4.2.2. *Y/B buildings, 0/1 strings and pairs of rabbits.*

Problem 11.4.1. Counting :

(1) buildings with yellow and blue floors, without consecutive yellow floors.
(2) strings of 0 and 1 without two consecutive 1.

The answer in both case is the Fibonacci sequence.

Those two problems are fundamentally the same :

There is a bijection between solutions of the first and solutions to the second !

So if you solve the one of them, you solve both of them.

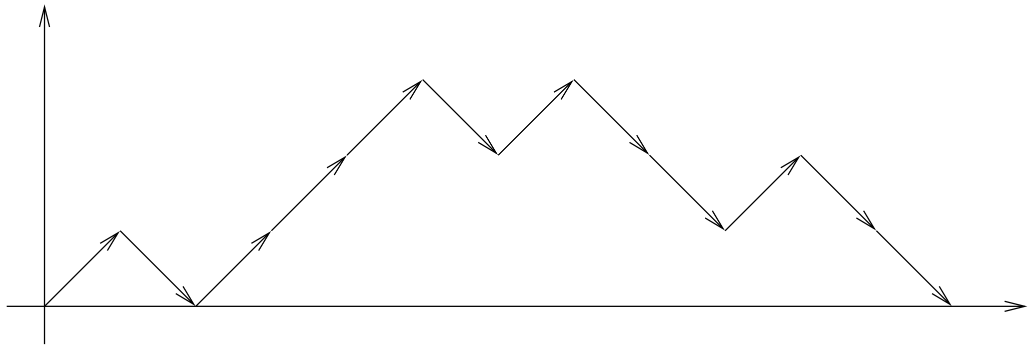
Problem 11.4.2. Counting pairs of rabbits after n generations (ex 29 p. 140).

Here also the solution is the Fibonacci sequence.

Can you find the bijection ?

11.4.2.3. *Strings of well-balanced parenthesis and Dyck paths.* A string of well-balanced parenthesis is a string such as `(()())`, where each open parenthesis is closed and vice versa.

Définition. A Dyck path of size n, is a path in $\mathbb{N} \times \mathbb{N}$ from $(0,0)$ to $(2n,0)$ such that each step is either $(1,1)$ or $(1,-1)$.



The important fact is that such a path never goes under the x-axis.

Exercice 44. Find all strings of well-balanced parenthesis of length 0, 2, 4,6, 8, 10.

Find all Dyck paths of size 0, 1, 2, 3, 4.

Do you notice something ?

Conjecture. *There are as many Dyck paths of size n than strings of well balanced parenthesis of length $2n$.*

How to prove this conjecture ?

Let D be the set of Dyck paths of size n.

Let S be the set of strings of well balanced parenthesis of length $2n$.

We just have to construct a bijection $f : S \to D$.

Take $s = s_1 \cdots s_{2n} \in S$, and build a path $f(s)$ in the following way :

read s from left to right ; if s_i is a `(`, go right and up, else go right and down.

Exercice 45. Draw $f(s)$ for the following strings : `()`, `()()`, `((()(()))()())`.

Why is $f(s)$ indeed a Dyck path of size n ?
- There are $2n$ steps to the right
- In s, there are as many $'('$ as $')'$, so the final position is $(2n, 0)$.
- At any position i in s, there are more $'('$ than $')'$ before i.

How to prove that f a bijection ?

Lets try to construct an inverse g for f :

Take a path p, and build a string $g(p)$ in the following way : go through p from left two right. For each up step, add a $'('$ to $g(p)$, and for each down step, add a $')'$.

Exercice 46. Apply g to the paths obtained in the previous exercises.

For the same reasons as above, the resulting string $g(p)$ is a string of well balanced parenthesis.

Moreover, by construction $f(g(p)) = p$ for any $p \in D$ and $g(f(s)) = s$, for any $s \in S$.

So g is indeed an inverse for f.

Therefore, f is a bijection and consequently $|S| = |D|$, as wanted.

11.4.2.4. *Ordered trees and strings of well-balanced parenthesis.*

Définition. The set of all ordered trees can be defined recursively as follow :

- A single node is an ordered tree
- If $t_1, \ldots, t_k$ are k ordered trees, the structure obtained by adding a common father to $t_1, \ldots, t_k$ is an ordered tree.

Exercice 47. Draw all ordered trees on $1, 2, 3, 4, 5$ nodes.

Exemple. Prove that any ordered tree on n nodes has $n - 1$ edges.

Ordered trees are defined recursively, so using recursion seems reasonable.

The property is true on a tree with one node.

Let t be a tree on $n > 1$ nodes.

Assume the property is true for any tree of size $< n$.

By construction, t is build by adding a common father to k trees $t_1, \ldots, t_k$.

Let n_i be the number of nodes of t_i.

We have $n = n_1 + \cdots + n_k + 1$.

Clearly $n_i < n$, so by induction each t_i has $n_i - 1$ edges.

Conclusion : the number of edges of t is :

$$(n_1-1)+\cdots+(n_k-1)+k=n_1+\cdots+n_k=n-1.$$

THÉORÈME. *There are as many trees with n nodes as strings of well balanced parenthesis of length $2(n-1)$.*

Let's define recursively a bijection f between trees and well balanced parenthesis :
– If t has a single node, then $f(t)$ is the empty string ;
– If t is build from k subtrees $t_1,\dots,t_k$, then $f(t) := (f(t_1))\cdots(f(t_k))$.

EXERCICE 48. Apply f to a few trees.

Here also, it's easy to construct the inverse of f, so f is a bijection.
Since f maps trees on n nodes on strings of length $2(n-1)$, and vice-versa, we are done with the proof of the theorem.

11.4.2.5. *Catalan Numbers.* We have seen that many different kind of objects (trees, strings of well balanced parenthesis, Dyck paths) are counted by the same sequence of numbers :

$$1,1,2,5,14,\dots$$

This sequence is called the Catalan sequence $C(0),C(1),C(2),C(3),\dots$.

PROBLEM 11.4.3. What is the value $C(n)$?

It's enough to count, for example, the number of Dyck paths of length $2n$.
There is a beautiful trick to do this. It's called *André's inversion principle.*
Will you find it ?

11.5. Conclusion

– There are often connections between apparently unrelated objects (trees, strings of well balanced parenthesis, Dyck paths, ...).
– Those connections, formalized by bijections, provide powerful methods for counting objects.
– The Sloane is very handy to suggest connections !

Cinquième partie

Modeling, Arithmetic, Computations and Languages

CHAPITRE 12

Groups, and other algebraic structures (section 8.1)

12.1. Introduction

A mathematical structure is a formal model which capture common properties or behavior of different sets.

It consists of an abstract set, together with operations and relations which obey certain rules.

For example the set of tasks to build a house, and the set of all courses available at CSM have something in common : the notion of prerequisite. This notion can be formalized using a structure of partially ordered set.

We will introduce here certain algebraic structures, which are useful to model arithmetic, as well as to the study of symmetries.

12.2. Permutations

Définition. Let A be a set. A *permutation* of A is a bijection from A to A.

Let S_A be the set of all permutations of A.

Let S_n be the set of all permutations of $\{1, \dots n\}$.

A permutation σ of S_n can be written in *array form* as follow :

$$\sigma := \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 4 & 5 & 3 & 6 \end{pmatrix},$$

meaning that $\sigma(1) = 2$, $\sigma(2) = 1$, $\sigma(3) = 4$, and so on.

Exercice 49. Write all permutations in S_1, S_2, and S_3.

What is the size of S_n ?

12.2.1. Operation on S_n.

S_n as a set is not very interesting.

Can we define operations on S_n ?

Proposition. *The composition $\bullet$is an operation of S_n.*

Indeed, the composition of two permutations is still a permutation.

We also call this operation product of permutations.

Exercice 50. Compute the following :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 5 & 1 & 6 & 3 & 4 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 4 & 1 & 2 \end{pmatrix} \bullet \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 4 & 2 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 4 & 1 & 2 \end{pmatrix} \bullet \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

The permutation above produces a *cycle* : $1 \to 2 \to 3 \to 4 \to 5 \to 1$

DÉFINITION. By (a, b, c, d), we denote the permutation σ such that $\sigma(a) = b$, $\sigma(b) = c$, $\sigma(c) = d$, $\sigma(d) = a$, and $\sigma(x) = x$ for any other element x.

Such a permutation is called a *cycle*.

A cycle (i, j) of length two is called a *transposition*.

A transposition $(i, i + 1)$ is called *elementary transposition*.

EXERCICE 51. Write $(3, 4, 2, 1)$ and $(2, 1, 3, 4)$ in array notation.

Write the following permutations in array notation :

$$(3, 4, 2, 1) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$(2, 1, 3, 4) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$(1, 2, 3) \bullet (4, 5) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

$$(4, 5) \bullet (1, 2, 3) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

$$(1, 2, 3) \bullet (3, 4) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

$$(3, 4) \bullet (1, 2, 3) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

PROPOSITION. *The product of two disjoint cycles is commutative.*

This is not the case for non-disjoint cycles.

EXERCICE 52. Are the following permutations cycles ?

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 4 & 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 4 & 5 & 3 & 6 \end{pmatrix}$$

PROBLEM 12.2.1. So there are permutations that are not cycles.

Would it be possible to write them using only cycles ?

THÉORÈME. *Any permutation can be written as a product of disjoint cycles.*

This product is unique, up to the order.

Exercice 53. Compute the following permutations

$$(2,3,1,5,4)^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 10 & 2 & 1 & 6 & 4 & 8 & 3 & 7 & 9 \end{pmatrix}^{131} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

Writting a permutation in cycle notation is a good way to compute powers of this permutation.

12.2.2. Generators of S_n. If you have a list, you can use a permutation to change its order.

Well, S_n is pretty big, so this makes quite a lot of possible operations.

However, some operations can be obtained by combining other operations.

Could we try to reduce the size of S_n ?

Définition. A subset G of S_n *generates* S_n if any permutation can be written as a product of elements of G.

Problem 12.2.2. Find a set of generators of S_n as small as possible.

Problem 12.2.3. Does the set of all cycles generate S_n ?

Problem 12.2.4. How many cycles are there ?

Can you do better ?

Problem 12.2.5. Does the set of all transpositions generate S_n ?

How many transpositions are there ?

Can you do better ?

Problem 12.2.6. Does the set of all elementary transposition generate S_n ?

How many elementary transpositions are there ?

By the way, how many elementary transpositions at most do you need to express a permutation ?

Can you do better than $n-1$ generators ?

12.3. Groups

12.3.1. Definitions and examples. The notation $[A, \bullet]$ denote a set A together with an operation $\bullet$."

Problem 12.3.1. Is there anything in common between $[S_n, \bullet]$ and $[\mathbb{Z}+]$.

Définition. Let A be a set with an operation $\bullet$

– The operation $\bullet$ is *associative* if :

$$(\forall x)(\forall y)(\forall z)\ x \bullet (y \bullet z) = (x \bullet y) \bullet z$$

– The operation $\bullet$ is *commutative* if :

$$(\forall x)(\forall y)\ x \bullet y = y \bullet x$$

– $i \in A$ is an *identity element* if :

$$(\forall x)\ x \bullet i = i \bullet x = x$$

– Assume $[A, \bullet]$ has an identity element i. Then $y \in A$ is an inverse for $x \in A$ if

$$y \bullet x = x \bullet y = i$$

What are the properties of $[\mathbb{Z}, +]$?

– $+$ associative ;
– $+$is commutative ;
– 0 is an identity element ;
– If x is an integer, then$-x$ is an inverse for x.

What are the properties of $[\mathbb{R}^*, \cdot]$?

Those structures have many common properties.

DÉFINITION. $[G, \bullet]$ is a *group* if G is a non empty set, and $\bullet$is a binary operation on G such that :

(1) $\bullet$ is associative
(2) an identity element exists (in G).
(3) each element of G has an inverse (in G) with respect to $\bullet$.

If the operation $\bullet$is commutative, G is called a *commutative group.*

EXERCICE 54. Which of the following are groups ?

(1) $[\mathbb{Z}, +]$
(2) $[\mathbb{N}, +]$
(3) $[\mathbb{Z}, -]$
(4) $[\{true, false\}, \wedge\}$
(5) $[\mathbb{R}, +]$
(6) $[\mathbb{R}, \cdot]$
(7) $[S_n, \bullet]$
(8) $[n\mathbb{Z}, +]$

What's the point ?

– Many structures, coming in particular from arithmetics, are groups.
– If you prove a property for groups in general, you have proved this property for all of those structures.
– Group theory is a vast field, and there are hundreds of theorems about groups waiting for you to apply them on your favorite structure.
– Basic tool to study the Rubicks Cube.

12.3.2. Just some examples of results and questions about groups.

THÉORÈME. *The identity of a group is unique.*

The inverse of an element is unique.

The equations $a \bullet x = b$ and $x \bullet a = b$ have a unique solution.

The inverse of a product can be computed by $(x \bullet y)^{-1} = y^{-1} \bullet x^{-1}$.

DÉFINITION. Let $[G, \bullet]$ be a group, and $A \subseteq G$.

Then $[A, \bullet]$ is a *subgroup* of $[G, \bullet]$ if $[A, \bullet]$ itself is a group.

Théorème. *The size of a subgroup divides the size of the group.*

Problem 12.3.2. Common problems about a group :

- Find a minimal set of generators
- Express an element of the group as product of generators
- Efficient computations
- Find all subgroups
- Are two groups structurally the same (are they isomorphic ?)

For playing with groups, there is a very nice program :

`http://www-history.mcs.st-and.ac.uk/~gap/`

12.3.3. Groups and Symmetries. One common use of groups if for the study of problems with symmetries.

Exemple. Computing the temperature in a house with a heater in the middle.

- Case 1 : the house has a random shape ;
- Case 2 : the house is round ;
- Case 3 : the house is square.

Most of the time, a problem with symmetry has a symmetric solution.

Using this can save a LOT of time.

Problem : finding the orbits

The set of all symmetries is a group, so group theory can help you.

12.3.4. Groups and isomorphy of graph.

Définition. The two following structures are labelled graphs :

If you exchange the nodes 3 and 4 of the first graph, you get the second.

The group S_n *acts* on graphs on n nodes.

Structurally, except for the numbering, there is no difference between them :

They are *isomorphic*.

Problem 12.3.3. Testing if two graphs are isomorphic.

This is an important but very difficult problem.

Définition. The following structure is an *unlabelled graph* :

It can be defined as an equivalence class of labelled graphs.

Problem 12.3.4. How many labelled graphs on n nodes ?

How many unlabelled graphs on n nodes ?

Group theory provides a tool (Pólya enumeration) for this.

12.4. Conclusion

- Groups are an abstract model for many common structures
- Results about groups apply to all of those structures
- Groups are a powerful tool to study problems with symmetries
- Groups often occurs when dealing with unlabeled structures

Bibliographie

[1] Judith L. Gersting. *Mathematical Structures for Computer Science*. W. H. Freeman and Company, 4 edition, 1998.

[2] Douglas R. Hofstadter. *Gödel, Escher, Bach : an eternal golden braid*. Basic Books Inc. Publishers, New York, 1979.

[3] G. Pólya. *How to solve it*. Princeton University Press, Princeton, NJ, second edition, 1988. A new aspect of mathematical method.